Faster Explicit-Trace Monitoring-Oriented Programming for Runtime Verification of Software Tests

KEVIN GUAN, Cornell University, USA
MARCELO D'AMORIM, North Carolina State University, USA
OWOLABI LEGUNSEN, Cornell University, USA

Runtime verification (RV) monitors program executions for conformance with formal specifications (specs). This paper concerns Monitoring-Oriented Programming (MOP), the only RV approach shown to scale to thousands of open-source GitHub projects when simultaneously monitoring passing unit tests against dozens of specs. Explicitly storing traces—sequences of spec-related program events—can make it easier to debug spec violations or to monitor tests against hyperproperties, which requires reasoning about sets of traces. But, most online MOP algorithms are *implicit trace*, *i.e.* they work event by event to avoid the time and space costs of storing traces. Yet, TRACEMOP, the only *explicit-trace* online MOP algorithm, is often too slow and often fails.

We propose LAZYMOP, a faster explicit-trace online MOP algorithm for RV of tests that is enabled by three simple optimizations. First, whereas all existing online MOP algorithms eagerly monitor *all* events as they occur, LAZYMOP lazily stores *only unique* traces at runtime and monitors them just before the test run ends. Lazy monitoring is inspired by a recent finding: 99.87% of traces during RV of tests are duplicates. Second, to speed up trace storage, LAZYMOP encodes events and their locations as integers, and amortizes the cost of looking up locations across events. Lastly, LAZYMOP only synchronizes accesses to its trace store after detecting multi-threading, unlike TRACEMOP's eager and wasteful synchronization of all accesses.

On 179 Java open-source projects, LAZYMOP is up to **4.9**x faster and uses **4.8**x less memory than TRACEMOP, finding the same traces (modulo test non-determinism) and violations. We show LAZYMOP's usefulness in the context of software evolution, where tests are re-run after each code change. LAZYMOP^e optimizes LAZYMOP in this context by generating fewer duplicate traces. Using unique traces from one code version, LAZYMOP^e finds all pairs of method m and spec s, where all traces for s in m are identical. Then, in a future version, LAZYMOP^e generates and monitors only one trace of s in m. LAZYMOP^e is up to **3.9**x faster than LAZYMOP and it speeds up two recent techniques that speed up RV during evolution by up to **4.6**x with no loss in violations.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: runtime verification, software testing

ACM Reference Format:
Kevin Guan, Marcelo d'Amorim, and Owolabi Legunsen. 2025. Faster Explicit-Trace Monitoring-Oriented Programming for Runtime Verification of Software Tests. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 405 (October 2025), 30 pages. https://doi.org/10.1145/3763183

1 Introduction

Runtime verification (RV) [41, 62, 74] monitors program executions against formal specifications (specs). Unless the context indicates otherwise, "RV" in this paper is Monitoring-Oriented Programming (MOP) [29], the only RV style shown to scale for simultaneously monitoring dozens of specs against *passing* tests in thousands of real-world projects [51, 67]. RV of such tests against

Authors' Contact Information: Kevin Guan, Cornell University, Ithaca, NY, USA, kzg5@cornell.edu; Marcelo d'Amorim, North Carolina State University, Raleigh, NC, USA, mdamori@ncsu.edu; Owolabi Legunsen, Cornell University, Ithaca, NY, USA, legunsen@cornell.edu.

ACM 2475-1421/2025/10-ART405
https://doi.org/10.1145/3763183

specs of correct usage protocols of JDK APIs helped find hundreds of confirmed bugs [77, 79, 92] because these (often multi-object) typestate [123] specs provide additional test oracles. Our results are limited to these kinds of specs, which were also used in these prior works on RV during testing.

MOP works in three main steps. First, a program is instrumented based on a given set of specs, so that spec-related events (*e.g.*, method calls and field accesses) are signaled at runtime. Second, monitors—usually automata—are dynamically synthesized to check if event sequences, *i.e.*, traces, satisfy the specs. Finally, monitors print warnings or perform error recovery if a spec is violated.

Most online MOP algorithms [26, 30, 31, 105] are *implicit-trace*: they work event by event without explicitly storing traces. Such algorithms optimize for RV of deployed systems, where the time and space costs to store traces are prohibitive. An exception is TraceMOP [53, 125], which stores traces but is often much slower than state-of-the-art (SoTA) implicit-trace JavaMOP [69, 87] and often fails (§4), making it hard to use in continuous integration (CI). Explicit-trace *offline* MOP [31] logs all events and checks them offline, but is often much slower and takes more space than TraceMOP.

We propose LazyMOP, a faster explicit-trace *online* MOP algorithm for RV during testing. LazyMOP can enable **six** important applications (A1-A6):

A1. *Speeding up RV during CI.* LazyMOP can enable new ways of using traces to speed up RV during CI. For example, we combine LazyMOP with existing evolution-aware techniques that speed up RV during CI by re-monitoring only specs related to code affected by changes [80, 82, 131].

A2. *Better violation debugging.* Violations take many hours to debug [77, 79, 92] mostly because current MOP algorithms only report the location of the last event in violating traces and developers must manually reconstruct violating traces. LazyMOP can enable semi-automated debugging for reasoning jointly about code and violating traces, and user studies on doing so.

A3. *Root causing violations.* Test non-determinism leads to different violations in multiple runs on the same *passing* tests and specs [52]. Such *flaky violations* make it hard to compare RV tools [51, 52], and cause false positives/negatives. Comparing traces helps root-cause flaky violations [53]. LazyMOP can enable new semi-automated ways to root-cause flaky violations.

A4. *Monitoring hyperproperties.* Non-functional, *e.g.*, security, policies are often hyperproperties [32]; monitoring them requires checking if *sets of traces* are in a theoretical "good" *set of sets of traces.* Many works on RV of hyperproperties [1, 16, 24, 42, 43, 56, 57, 96] were evaluated offline. LazyMOP can enable new research on online monitoring of hyperproperties during CI.

A5. *Test-suite reduction (TSR) for RV.* No prior work speeds up RV via TSR [70, 110, 111, 113, 116, 130, 135], which speeds up testing by minimizing the redundancy among tests—two tests satisfying the same test requirements (*e.g.*, covered statements or killed mutants) are redundant with each other. LazyMOP can help specialize TSR for RV, by using traces as test requirements.

A6. *Trace-guided test generation.* RV can only monitor executed program paths. LazyMOP can enable new techniques that use previously-seen traces as feedback to drive automated test generation to previously-uncovered paths that are more likely to signal spec-related events.

LazyMOP aims to perform explicit-trace RV of tests at near implicit-trace JavaMOP speeds. To do so, LazyMOP embodies four simple optimizations, three of which are new in this paper:

1. **Lazy monitoring**. LazyMOP collects unique traces during execution and monitors each unique trace *once*, just before program termination. Lazy monitoring speeds up explicit-trace RV by exploiting the high repetitiveness and wastefulness of RV during testing—99.87% of monitored traces cannot find new bugs: they are duplicates of the other 0.13% [51]. Lazy monitoring is for testing. In deployment, it is crucial to eagerly monitor events as soon as they are signaled [3, 41].
2. **Compact trace store**. LazyMOP adapts TraceMOP's trie-like data structure to only store unique traces; it (i) saves space by storing shared trace prefixes once; (ii) tracks each event's parameters; and (iii) adapts a trace-slicing algorithm [26] to find the right prefix for new events.

```
BAOS(ByteArrayOutputStream b, OutputStream o) {
  creation event init after(ByteArrayOutputStream b) returning(OutputStream o) :
    call(OutputStream+.new(..)) && args(b, ..) {}
  event write before(OutputStream o) : call(* OutputStream+.write*(..)) && target(o) {}
  event flush before(OutputStream o) : call(* OutputStream+.flush(..)) && target(o) {}
  event close before(OutputStream o) : call(* OutputStream+.close(..)) && target(o) {}
  event tobytearray before(ByteArrayOutputStream b):call(* ByteArrayOutputStream+.toByteArray(..))&&target(b){}
  event tostring before(ByteArrayOutputStream b) : call(* ByteArrayOutputStream+.toString(..)) && target(b) {}
  fsm : // see Fig 3
  @fail {/*print violation*/)} }
```

Fig. 1. The ByteArrayOutputStream_FlushBeforeRetrieve (BAOS) spec, written in an AspectJ-based DSL.

3. **Event encoding**. For fast trace store access, LazyMOP encodes each event and its location as an integer that is decoded before lazy monitoring. So, each LazyMOP "event" is a (name, location) pair. LazyMOP also amortizes the cost of location look-up across all events at a location.
4. **On-demand synchronization**. LazyMOP synchronizes trace-store accesses only after witnessing multi-threading in monitored code. Locking RV data structures induces non-trivial overheads [51], but avoiding those costs during eager monitoring is hard and requires potentially costly sound static analysis of the monitored code or specializing RV per program.

We compare LazyMOP's overheads and violations with those of TraceMOP, the SoTA explicit-trace online MOP algorithm. On 179 open-source projects, LazyMOP is up to 4.9x (avg. 1.9x) faster and uses up to 4.8x (avg. 1.6x) less peak memory than TraceMOP, finding the same traces (modulo test non-determinism) and violations. Also, LazyMOP finishes in 18 projects where TraceMOP does not. Surprisingly, LazyMOP even outperforms JavaMOP on 33 projects where the cost of repeatedly and eagerly monitoring duplicate traces dominates JavaMOP's time.

To begin showing LazyMOP's usefulness, we also propose LazyMOP^e to speed up RV as software evolves, towards A1 above. LazyMOP^e exploits a finding from a formative study that we conduct (§3.2.1): *all* events in many duplicate traces are from the same method. On top of savings that LazyMOP obtains by reducing the cost of *monitoring* duplicate traces, LazyMOP^e aims to also reduce the costs of *generating* duplicate traces. So, in one code version, LazyMOP^e finds all (method m, spec s) pairs where duplicate traces for s in m are identical and have only events in m. Then, LazyMOP^e transforms all such m so only one trace of s is generated in m in the next version. §3.2 discusses how LazyMOP^e incrementally detects and maintains the set of (m, s) as code evolves.

We measure LazyMOP^e's overhead reduction and empirically evaluate its safety (*i.e.*, the ability to find all new violations after code changes) in two ways, using 2,401 versions of 104 projects. First, we *compare* LazyMOP^e with two implicit-trace evolution-aware RV techniques—$\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ [82, 131]. LazyMOP^e is up to 3.1x faster than, and as safe as $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$. Second, we combine these techniques with LazyMOP^e, yielding speedups of up to 4.6x without making them less safe.

This paper makes the following contributions.

★ **Optimizations.** LazyMOP's three simple optimizations speed up explicit-trace RV of tests.
★ **Characterizing Duplicate Traces.** We study duplicate traces, to better understand how they manifest in real-world projects and how to exploit them for faster RV during software evolution.
★ **Combination.** Based on a study finding, LazyMOP^e combines LazyMOP with targeted disabling of monitoring during CI when all events in duplicated traces are in the same method.

LazyMOP, LazyMOP^e, and all our artifacts are at https://github.com/SoftEngResearch/lazymop.

2 Background

First, §2.1 illustrates MOP via (i) a spec of correct Java API usage; (ii) how implicit-trace JavaMOP works; (iii) some JavaMOP limitations that motivate explicit-trace RV; and (iv) how explicit-trace TraceMOP works. Then, §2.2 dives deeper into the MOP algorithms in JavaMOP and TraceMOP.

```
1  public class TestSerializable extends TestCase {
2    public static Object cloneViaSerialization (Serializable obj) throws IOException, ClassNotFoundException {
3      ByteArrayOutputStream baos = new ByteArrayOutputStream ();
4      ObjectOutputStream oos = new ObjectOutputStream (baos); /*INSTR: BAOS.init*/
5      /*INSTR: BAOS.write*/ oos.writeObject (obj);
6      ByteArrayInputStream bias = new ByteArrayInputStream (/*INSTR: BAOS.tobytearray*/ baos.toByteArray ());
7      ObjectInputStream ois = new ObjectInputStream (bias);
8      return ois.readObject (); }
9  }
10 public class TestAlphabet extends TestCase {
11   public void testReadResolve () throws IOException, ClassNotFoundException {...
12     Alphabet dict2 = (Alphabet) TestSerializable.cloneViaSerialization (dict); ...}
13 }
14 public class TestLabelsSequence extends TestCase {
15   public void testSerializable () throws IOException, ClassNotFoundException {...
16     LabelsSequence lblseq2 = (LabelsSequence) TestSerializable.cloneViaSerialization (lblseq); ...}}
```

Fig. 2. An example of instrumented code and some of its unit tests.

2.1 MOP by Examples

A MOP spec. Fig 1 shows the `ByteArrayOutputStream_FlushBeforeRetrieve` (BAOS) spec; it is violated if an `OutputStream` instance os that is built on a `ByteArrayOutputStream` instance baos calls `baos.toByteArray` or `baos.toString` before closing os, possibly reading incomplete data. BAOS helped find several bugs that passing tests alone missed [77, 79]. BAOS and scores of other specs were formalized from natural language in the Javadoc of JDK APIs [76, 87]. These specs are essentially (multi-object) typestate protocols expressed in past- and future-time linear temporal logic (LTL), extended regular expressions (ERE), finite-state machines (FSM), etc.

Like BAOS, all specs in this paper consist of event definitions (lines 2–8), a formal safety property defined over events (line 9), and a handler that is triggered if the property is violated (line 10).

Events and their parameters. Each event definition has a name and a list of parameters that it binds. For example, lines 2–3 define event "`init`"; each time it is signaled at runtime, its parameters are the `ByteArrayOutputStream` argument and the returned `OutputStream`, corresponding to the newly created object of that type. The union of parameters in all event definitions forms a spec's parameters (line 1). Parameters are important in a MOP spec because the safety property is universally quantified. For example, line 1 in Fig 1 means that MOP must check the traces related to BAOS *for all* pairs of `OutputStream` and `ByteArrayOutputStream` related by `init`. In addition to `init`, BAOS defines five other events: (i) `write` captures calls to `o.write*()` (methods whose names start with `write`); (ii) `flush` captures calls to `o.flush()`; (iii) `close` captures calls to `o.close()`; (iv) `tobytearray` captures calls to `b.toByteArray()`; and (v) `tostring` captures calls to `b.toString()`.

Properties and handlers. Line 9 describes BAOS's safety property as an FSM shown in Fig 3; it is violated if a monitor sees an event for which no transition exists from the current state. It is hard to check violations of BAOS with testing alone. RV triggers the handler if a trace violates this property. Handlers can be any user-provided code, *e.g.*, for error-recovery when RV is used in deployment. For testing, handlers simply print messages to help debug violations.

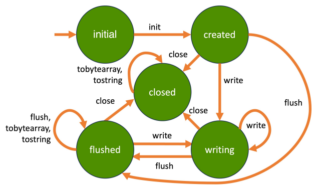

Fig. 3. BAOS's FSM; its violating state (not shown) is reached on seeing an event with no transition from the current state.

Implicit-trace RV with JavaMOP. Fig 2 shows simplified code from project `mimno/Mallet`. To signal relevant BAOS events at runtime, JavaMOP first instruments the code based on BAOS's event definitions. The `after` keyword in BAOS's `init` event definition (Fig 1, line 2) causes JavaMOP-instrumented code to signal an `init` event *after* calling "`new OutputStream(baos)`" (Fig 2, line 4). Similarly, the `before` keyword (Fig 1, line 4) causes JavaMOP-instrumented code to signal *before* calling `oos.writeObject(obj)` (Fig 2, line 5), and line 7 in Fig 1 causes JavaMOP-instrumented code to signal *before* calling `baos.toByteArray()` (Fig 2, line 6). The `creation` keyword on line 2 in Fig 1 causes JavaMOP to synthesize a new BAOS monitor when each test calls line 4 in Fig 2.

Four subtle points are essential to understanding MOP. First, we can see that JavaMOP should create two monitors in Fig 2. Monitor m_1 sees trace $\tau_1 = \texttt{init}\langle\texttt{baos}_1, \texttt{os}_1\rangle \rightsquigarrow \texttt{write}\langle\texttt{os}_1\rangle \rightsquigarrow \texttt{tobytearray}\langle\texttt{baos}_1\rangle$, and m_2 sees $\tau_2 = \texttt{init}\langle\texttt{baos}_2, \texttt{os}_2\rangle \rightsquigarrow \texttt{write}\langle\texttt{os}_2\rangle \rightsquigarrow \texttt{tobytearray}\langle\texttt{baos}_2\rangle$[1]. But, JavaMOP's online MOP algorithm cannot see this fact *a priori*, since it eagerly processes each event as it arises. In fact, with multi-threading, events in τ_1 and τ_2 could be signaled in an interleaving manner. MOP uses a *parametric trace slicing* algorithm [26, 31] (§2.2) to decide when to create monitors and to correctly dispatch signaled events to them, based on event parameters.

Second, for efficiency reasons, JavaMOP synthesizes monitors (e.g., Fig 3) with only event names in their alphabets (in the automata-theoretic sense). But, elements in τ_1 and τ_2 are event names plus parameter instances. In MOP, τ_1 and τ_2 are called *parametric traces* (§2.2). Parametric trace slicing also "forgets" parameter instances to obtain *non-parametric traces* that each monitor sees.

Third, JavaMOP does not store traces that monitors see. Rather, monitors transition to a next (potentially violating) state after seeing an event, then discards that event. So, we say JavaMOP performs implicit-trace RV. In fact, although most online RV algorithms that we know are also implicit-trace [10, 28, 34, 87], they often incur high overheads. So, avoiding the time and space costs to explicitly track monitored traces—which can be up to billions per project [82]—is conventionally seen as prudent, especially when using RV to monitor deployed systems.

Finally, if the state of a parameter instance (e.g., $\texttt{baos}_1, \texttt{os}_2$) is essential to determining spec violations, then (i) different event definitions (and names) can be used to track when parameter instances' states change at runtime; or (ii) that state change is included in what it means for an event to be triggered. In either case, for all MOP specs in this paper, it is sufficient to only reason about names of signaled events when checking for violations. For example, BAOS is not concerned with the state of its parameters (the `..` in its event definitions mean "don't care").

Some JavaMOP Limitations that Motivate Explicit-trace RV. Implicit-trace RV algorithms cannot enable the applications that motivate explicit-trace RV (*e.g.*, A1–A6 in §1). We discuss limitations of implicit-trace JavaMOP that motivate two of those applications here.

JavaMOP-reported violations take many person hours to debug [77, 79, 92]. For example, we can see that τ_1 and τ_2 violate BAOS: there is no transition on `tobytearray` out of the `writing` state in Fig 3. Yet, when monitoring `testReadResolve` and `testReadResolve` (both call `cloneViaSerialization`) against BAOS in Fig 2, the only feedback a user gets from JavaMOP is a message like *"BAOS was violated on line 6"*. That is, JavaMOP only reports the location of the last event in a violating trace. Users must then (i) manually reconstruct the trace that monitors see, including for events in third-party library binaries; and then (ii) reason jointly about the spec and the reconstructed trace.

Definition 1. Two traces for a spec are identical if they are the same sequence of non-parametric events and the event at each position in both traces is signaled at the same code location. If multiple monitors observe identical traces, those traces are *repeatedly and identically monitored.*

During RV of tests, repeatedly monitored identical traces are generated in paths that are in loops [51, 98], or methods that are called multiple times [51]. But, there is no bug-finding benefit to monitoring identical traces more than once. For example, due to "forgetting" parameter instances during trace slicing, m_1 and m_2 eventually see identical traces: τ_1 and τ_2 become $\tau = \texttt{init:4} \rightsquigarrow \texttt{write:5} \rightsquigarrow \texttt{tobytearray:6}$, where integers after colons are line numbers where events are signaled. Since m_1 and m_2 see identical traces, only one of them is necessary to find the BAOS violation. In fact, many similar tests exist in `mimno/Mallet` that produce τ.

[1]For simplicity, we use subscripts to uniquely identify parameter instances; JavaMOP uses weak references [87].

Implicit-trace JavaMOP does not track event locations or traces, so it cannot determine if traces are identical. LazyMOP aims to speed up explicit-trace MOP-style RV by exploiting findings that (i) 99.87% of traces are identical during RV of tests; and (ii) monitoring overhead is often dominated by the cost of repeatedly monitoring identical traces *hundreds of millions of times* [51].

Limitation and Scope. Definition 1 and our claim about the wastefulness of repeatedly monitoring identical traces is limited to (i) our specs of correct Java API usage protocols, so it may not apply to other kinds of specs, like the absence of data races; and (ii) monitoring those specs to find more bugs during testing—in deployment-time RV, repeatedly monitoring identical traces is essential.

Explicit-trace RV with TraceMOP. TraceMOP extends JavaMOP by storing traces that monitors see in a store that is shared by all monitors for all specs [53]. TraceMOP's shared trace store improves over Guan and Legunsen's naïve approach, which simply adds a list field to each monitor object [51]. That naïve approach often does not scale well because (i) it is not possible to predict the length of each trace, so for monitors that see tens of thousands of events, frequent resizing of enhanced monitors' list fields is very costly; and (ii) separately tracking hundreds of millions of repeatedly monitored identical traces easily overwhelms memory or disk storage. §2.2 discusses how TraceMOP's trace store works with JavaMOP's monitor indexing tree. Although TraceMOP's trace store is more efficient, and succeeds on more projects, than the naïve approach, TraceMOP is still up to 10.3x, or 7.1 hours slower than JavaMOP, and crashes on several projects. These performance limits of TraceMOP, whose trace store already exploits prior work's finding on dominance of repeatedly monitored identical traces, suggest the need for new and faster explicit-trace MOP-style RV algorithms. LazyMOP is such an algorithm; its optimizations target the main sources of inefficiency that we observe from profiling TraceMOP (Fig 8).

2.2 A Quick Algorithmic Tour of MOP and its Incarnation in JavaMOP and TraceMOP

Preliminaries. A *trace* is a sequence of events [102]. A *parametric event* has abstract parameters that are bound to heap objects at runtime. A *parametric trace* contains parametric events. A spec encodes a *property* that maps traces to categories (*e.g.*, `violating`, `not-violating`). In this paper, specs encode *parametric properties* that map parametric traces to categories. Consider parametric trace $\tau_3 = \texttt{init}(\texttt{baos}_3, \texttt{os}_3) \rightsquigarrow \texttt{write}(\texttt{os}_3) \rightsquigarrow \texttt{flush}(\texttt{os}_4) \rightsquigarrow \texttt{tobytearray}(\texttt{baos}_4) \rightsquigarrow \texttt{tobytearray}(\texttt{baos}_3)$. Naïvely ignoring the parameters yields non-parametric trace $\tau_3' = \texttt{init} \rightsquigarrow \texttt{write} \rightsquigarrow \texttt{flush} \rightsquigarrow \texttt{tobytearray} \rightsquigarrow \texttt{tobytearray}$, for which the BAOS monitor in Fig 3 wrongly fails to find a violation. MOP uses *parametric trace slicing* [26, 31], defined shortly, to correctly handle parametric traces.

A *parameter instance* θ is a partial function from spec parameter types to objects. So, in τ_3, the `init` event's parameter instance is $\theta_1 = \langle \texttt{ByteArrayOutputStream} \longrightarrow \texttt{baos}_3, \texttt{OutputStream} \longrightarrow \texttt{os}_3 \rangle$, or $\theta_1 = \langle \texttt{baos}_3, \texttt{os}_3 \rangle$ for short. Similarly, the parameter instance of the `flush` event in τ_3 is $\theta_2 = \langle \texttt{os}_4 \rangle$; it binds no object to the `ByteArrayOutputStream` type. Lastly, the parameter instance of the last `tobytearray` event in τ_3 is $\theta_3 = \langle \texttt{baos}_3 \rangle$; it binds no object to `OutputStream`.

Next, we repeat definitions from [26, 30, 31, 105] verbatim, to help explain MOP and trace slicing.

Definition 2. Parameter instances θ and θ' are **compatible** if for any parameter $x \in \texttt{Dom}(\theta) \cap \texttt{Dom}(\theta')$, $\theta(x) = \theta'(x)$, where $\texttt{Dom}(\theta)$ denotes θ's domain. If θ and θ' are compatible, we can **combine** them, written $\theta \sqcup \theta'$, as follows:

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{when } \theta(x) \text{ is defined} \\ \theta'(x) & \text{when } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

θ' is **less informative** than θ, denoted as $\theta' \sqsubseteq \theta$, iff. for any $x \in X$, where X is a set of parameter types, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$. $\sqsubseteq$ is a partial order.

In τ_3, θ_1 is not compatible with θ_2: they "disagree" on the object bound to `OutputStream`. But, θ_1 is compatible with θ_3 as they "agree" on parameter $\texttt{baos}_3$ that they bind to `ByteArrayOutputStream`.

Definition 3. For parametric trace τ and parameter instance θ, the **θ-trace slice** $\tau\upharpoonright_\theta$ is a non-parametric trace defined recursively as:

- $\epsilon \upharpoonright_\theta = \epsilon$, where ϵ is the empty trace/word, and
- $(\tau\, e\langle\theta'\rangle) \upharpoonright_\theta = \begin{cases} (\tau \upharpoonright_\theta)\, e & \text{when } \theta' \text{ is less informative than } \theta\text{, i.e., } \theta' \sqsubseteq \theta \\ \tau \upharpoonright_\theta & \text{when } \theta' \text{ is } \textit{not} \text{ less informative than } \theta\text{, i.e., } \theta' \not\sqsubseteq \theta \end{cases}$

A *trace slice* is a projection of a parametric trace onto a set of compatible parameter instances. *Trace slicing* decomposes a parametric trace τ into non-parametric slices for each compatible set of parameter instances θ and their combinations. Trace slice $\tau\upharpoonright_\theta$ first projects τ onto θ, discarding events whose parameters are not compatible with θ. Then, $(\tau\upharpoonright_\theta)\, e$ "forgets" the parameters to obtain a non-parametric trace slice—a sequence of event names (with no parameters)—so that monitors do not spend time reasoning about parameters. Applying Definition 3 to τ_3 yields two slices τ_3^1 = `init` $\rightsquigarrow$ `write` $\rightsquigarrow$ `tobytearray`, which violates a BAOS monitor, and τ_3^2 = `flush` $\rightsquigarrow$ `tobytearray` for which no `init` event triggers BAOS monitor creation from $\texttt{baos}_4$ and $\texttt{os}_4$.

JavaMOP's Monitoring Algorithm. We first use an abstract example to illustrate JavaMOP's event-by-event online algorithm $\mathbb{D}\langle X\rangle$ (Algorithm 1) and motivate its technical details that we present next. Say the only parameter instance seen so far is $\theta_a = \langle a_1\rangle$, whose slice $e_1 \rightsquigarrow e_2$ was checked by monitor m_1. If event $e_3(\theta_b = \langle a_1, b_1\rangle)$ arrives next, $\mathbb{D}\langle X\rangle$ clones monitor m_1 in its current state to check θ_b's slice, $e_1 \rightsquigarrow e_2 \rightsquigarrow e_3$. The reason is that θ_b is more informative than θ_a, and θ_a is the most informative compatible instance seen so far that is less informative than θ_b. $\mathbb{D}\langle X\rangle$ tracks m_1's and m_2's states, in case future events' parameter instances are more informative than θ_a or θ_b. Say event $e_4(\theta_c = \langle a_1, b_2\rangle)$ arrives next. θ_c is more informative than θ_a but not θ_b. So, $\mathbb{D}\langle X\rangle$ clones m_3 from m_1 to check θ_c's slice, $e_1 \rightsquigarrow e_2 \rightsquigarrow e_4$. If event $e_5(\theta_d = \langle a_1, b_1, c_1\rangle)$ arrives next, the most informative compatible parameter instance seen so far that is less informative than θ_d is θ_b. So, $\mathbb{D}\langle X\rangle$ clones monitor m_4 from m_2 to check θ_d's slice: $e_1 \rightsquigarrow e_2 \rightsquigarrow e_3 \rightsquigarrow e_5$.

Remark. For simplicity, we follow prior work and only describe the core of all MOP algorithms in this paper, leaving out many implementation details. In particular, we only present how each algorithm handles an event and leave out important details on spec compilation, instrumentation, simultaneous monitoring of specs, monitor garbage collection, etc. [23, 29, 68, 69, 87, 99]. So, despite their relative simplicity in our presentation, implementations of each algorithm in JavaMOP, TraceMOP, and LazyMOP involve thousands of lines of Java code.

The inputs to $\mathbb{D}\langle X\rangle$ (Algorithm 1) are a *property parameter enable set* [26] $\text{enable}_{\mathcal{G}}^{X}$ (computed once per spec at compile time) and a template M (*e.g.*, a Java class definition) for synthesizing monitors for a spec. Intuitively, $\text{enable}_{\mathcal{G}}^{X}$ allows $\mathbb{D}\langle X\rangle$ to avoid creating useless monitors whose slice cannot reach a violating state. Such monitors only consume memory and slow down RV. In practice, $\text{enable}_{\mathcal{G}}^{X}$ is a map from each event name e to a set of sets of parameter types, each of which denotes parameters that must have been seen before e occurs for a monitor to be able to reach a violating state. The $\text{enable}_{\mathcal{G}}^{X}$ for BAOS maps all its six event names to the set of sets $\{\{b, o\}\}$ where `b` is a `ByteArrayOutputStream`, and `o` is an `OutputStream`.

$\mathbb{D}\langle X\rangle$ uses several globals for efficiency or soundness. Δ maps each parameter instance $\theta \equiv [X \rightharpoonup V]$ to the state S of the monitor checking its slice. $\mathcal{U}$ maps each encountered instance θ to the set of instances seen so far that are more informative than θ. The range of $\mathcal{U}$ is a set of sets, hence the $\mathcal{P}_f$ (powerset) notation. It is more efficient for $\mathbb{D}\langle X\rangle$ to look up more informative instances in $\mathcal{U}$ than to recompute that set per event. The integer timestamp is used in $\mathcal{T}$ to track when a monitor m_a is created due to a creation event. Monitors that are later cloned from m_a get the same timestamp as m_a in $\mathcal{T}$. Lastly, $\text{disable}(\theta)$ maps instance θ to the last time a monitor for θ processed

Algorithm 1 JavaMOP's implicit-trace monitoring algorithm, called $\mathbb{D}\langle X\rangle$ [26, 27].

```
Inputs: $\text{enable}^X_{\mathcal{G}}$: $[\mathcal{E} \rightharpoonup \mathcal{P}_f(\mathcal{P}_f(X))]$, $M$ : a monitor template
Globals: $\Delta : [[X \rightharpoonup V] \rightharpoonup S]$, $\mathcal{U} : [X \rightharpoonup V] \to \mathcal{P}_f([X \rightharpoonup V])$, timestamp : integer, $\mathcal{T} : [[X \rightharpoonup V] \rightharpoonup \text{integer}]$,
        disable: $[[X \rightharpoonup V] \rightharpoonup \text{integer}]$
Initialization: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta$, timestamp $\leftarrow 0$
 1: procedure main($e\langle\theta\rangle$)
 2:   if $\Delta(\theta)$ undefined then
 3:     createNewMonitorStates($e\langle\theta\rangle$)
 4:     if $\Delta(\theta)$ undefined and $e$ is a creation event then defineNew($\theta$)
 5:     disable($\theta$) $\leftarrow$ timestamp; timestamp $\leftarrow$ timestamp + 1
 6:   for all $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ s.t. $\Delta(\theta')$ defined do $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
 7: procedure createNewMonitorStates($e\langle\theta\rangle$)
 8:   for all $X_e \in \text{enable}^X_{\mathcal{G}}(e)$(in reversed topological order) do
 9:     if Dom($\theta$) $\not\subseteq X_e$ then
10:       $\theta_m \leftarrow \theta'$ s.t. $\theta' \sqsubset \theta$ and Dom($\theta'$) = Dom($\theta$) $\cap X_e$
11:       for all $\theta'' \in \mathcal{U}(\theta_m) \cup \{\theta_m\}$ s.t. Dom($\theta''$) = $X_e$ do
12:         if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then defineTo($\theta'' \sqcup \theta, \theta''$)
13: procedure defineNew($\theta$)
14:   for all $\theta'' \sqsubset \theta$ do if $\Delta(\theta'')$ defined then return
15:   $\Delta(\theta) \leftarrow \iota$; $\mathcal{T}(\theta) \leftarrow$ timestamp; timestamp $\leftarrow$ timestamp + 1
16:   for all $\theta'' \sqsubset \theta$ do $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
17: procedure defineTo($\theta, \theta'$)
18:   for all $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ do if disable($\theta''$) > $\mathcal{T}(\theta')$ or $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$ then return
19:   $\Delta(\theta) \leftarrow \Delta(\theta')$; $\mathcal{T}(\theta) \leftarrow \mathcal{T}(\theta')$; for all $\theta'' \sqsubset \theta$ do $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
```

an event. $\mathbb{D}\langle X\rangle$ uses $\mathcal{T}$ and disable to ensure soundness of its enable-set optimization, namely that instances for which $\mathbb{D}\langle X\rangle$ skips monitor creation or cloning have non-violating slices.

The instrumented code and tests signal spec-related event $e\langle\theta\rangle$ by invoking $\mathbb{D}\langle X\rangle$'s `main` procedure (lines 1–6). There, line 2 first checks if there is no monitor for θ's trace slice in Δ. If so, line 3 invokes createNewMonitorStates to potentially clone monitors (using defineTo on lines 17–19) for newly encountered instances that arise from combining θ with previously seen instances that are less informative than θ. If $\Delta(\theta)$ is still undefined after createNewMonitorStates returns and e is a creation event, then line 4 invokes defineNew, defined on lines 13–16, to assign a new monitor in its initial state ι to $\Delta(\theta)$. Finally, on line 6, the `main` procedure updates states of monitors for θ and all instances that are more informative than θ, *i.e.*, $\mathcal{U}(\theta)$, according to e and their transition functions σ. A violation is reported if a monitor transitions to an error state.

To save space, we elide line-by-line explanation of $\mathbb{D}\langle X\rangle$'s trace-slicing algorithm in `createNewMonitorStates`, `defineNew`, and `defineTo`. Those details and $\mathbb{D}\langle X\rangle$'s correctness proofs are in [26, 27].

How TraceMOP Extends $\mathbb{D}\langle X\rangle$ for Explicit-trace RV. Algorithm 2 shows at a high level how TraceMOP extends $\mathbb{D}\langle X\rangle$ to track traces. TraceMOP adds two new globals: a trie-like data structure (`trie`) and a map from parameter instances to trie nodes ($\mathcal{B}$). Just before a program terminates, TraceMOP algorithm's `main` procedure uses $\mathcal{B}$ to traverse `trie`, producing all traces (lines 2–4). Otherwise, the processEvent($e\langle\theta\rangle, l$) procedure is invoked, which differs from $\mathbb{D}\langle X\rangle$.processEvent in three ways: (i) it takes an additional argument l for the line of code where $e\langle\theta\rangle$ is signaled; (ii) it uses createNewMonitorStates′ and defineNew′ instead of createNewMonitorStates and defineNew, respectively; and (iii) it tracks traces in `trie` as sequences of (e, l) pairs (line 8). defineNew′ first invokes

Algorithm 2 TraceMOP's explicit-trace monitoring algorithm [53], which extends $\mathbb{D}\langle X\rangle$.

```
Inputs: same as $\mathbb{D}\langle X\rangle$, Outputs: traces : set of unique traces
Globals: all globals in $\mathbb{D}\langle X\rangle$, lock : a lock, trie : a trie-like data structure, $\mathcal{B}$ : $[[X \rightharpoonup V] \rightharpoonup$ trie node]
Initialization: $\mathbb{D}\langle X\rangle$'s initialization, trie $\leftarrow$ a root trie node
 1: procedure main($e\langle\theta\rangle, l$) :
 2:   if $e$ is termination signal then
 3:     traces $\leftarrow \{\}$; for all $\theta \in$ Dom($\mathcal{B}$) do $\mathcal{B}(\theta)$.monitors $\leftarrow \mathcal{B}(\theta)$.monitors + 1;
 4:     for all n $\in$ trie.nodes s.t. |n.monitors| > 0 do { $\tau \leftarrow$ path(trie.root, n); traces $\leftarrow$ traces $\cup \{\tau\}$}; return
 5:   lock.acquire(); processEvent($e\langle\theta\rangle, l$); lock.release()
 6: procedure processEvent($e\langle\theta\rangle, l$) :
 7:   // Same steps as in $\mathbb{D}\langle X\rangle$.main($e\langle\theta\rangle$), but uses createNewMonitorStates′ and defineNew′ instead.
 8:   for all $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ s.t. $\Delta(\theta')$ defined do $n \leftarrow \mathcal{B}(\theta')$.next($e + l$); $\mathcal{B}(\theta') \leftarrow n$
 9: procedure createNewMonitorStates′($e\langle\theta\rangle$) :
10:   // Same steps as in $\mathbb{D}\langle X\rangle$.createNewMonitorStates($e\langle\theta\rangle$), but uses defineTo′ instead.
11: procedure defineNew′($\theta$) : $\mathbb{D}\langle X\rangle$.defineNew($\theta$); $\mathcal{B}(\theta) \leftarrow$ trie.root
12: procedure defineTo′($\theta, \theta'$) : $\mathbb{D}\langle X\rangle$.defineTo($\theta$); $\mathcal{B}(\theta) \leftarrow \mathcal{B}(\theta')$
```

$\mathbb{D}\langle X\rangle$.defineNew, then updates $\mathcal{B}$ by pointing the monitor for θ's slice to `trie`'s root after creating a new monitor (line 11). Similarly, defineTo′ first invokes $\mathbb{D}\langle X\rangle$.defineTo, then updates $\mathcal{B}$ when a monitor is cloned (line 12). By re-using all steps in $\mathbb{D}\langle X\rangle$'s `processEvent`, `createNewMonitorStates`, `defineNew`, and `defineTo` procedures, TraceMOP merely adds trace-tracking features to JavaMOP's monitoring algorithm.

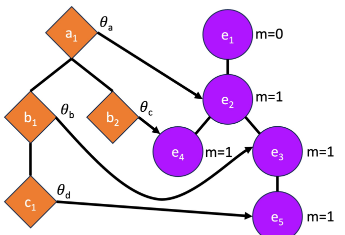

Fig. 4. TraceMOP's indexing tree (left) and `trie` (right) for our simple running example.

To speed up trace slicing, JavaMOP uses an indexing tree data structure to look up monitors for parameter instances. The left side of Fig. 4 (in orange) is a simplified view of that tree after observing all five events (e_1, e_2, e_3, e_4, and e_5) in our abstract example above. The right side (in purple) shows TraceMOP's `trie` for tracking trace slices seen so far. Note that TraceMOP's indexing tree is *only* used for monitor look-up; it does not interact with `trie`.

3 Techniques

3.1 LazyMOP

We aim to make LazyMOP work at near JavaMOP speeds, while explicitly tracking traces, by reducing the costs of repeatedly monitoring identical traces. LazyMOP's design is shown in Algorithm 3, which we call $\mathbb{E}\langle X\rangle$. We next describe $\mathbb{E}\langle X\rangle$ and its simple optimizations, and contrast $\mathbb{E}\langle X\rangle$ with $\mathbb{D}\langle X\rangle$ and the algorithm in TraceMOP, the SoTA explicit-trace MOP-style RV technique.

LazyMOP takes the same inputs as $\mathbb{D}\langle X\rangle$ and outputs all unique traces T and (for convenience) T_v, the subset of T that violates a spec. Globals $\mathcal{T}$, $\mathcal{U}$, disable, and timestamp in $\mathbb{E}\langle X\rangle$ are the same as in $\mathbb{D}\langle X\rangle$ and globals trie and lock in $\mathbb{E}\langle X\rangle$ are the same as in TraceMOP (Algorithm 2). But, $\mathbb{E}\langle X\rangle$ uses $\mathcal{D}$ instead of $\mathbb{D}\langle X\rangle$'s Δ. $\mathcal{D}$ maps each parameter instance $\theta \equiv [X \rightharpoonup V]$ to a node n in `trie` such that nodes on the path from trie.root to n represent θ's slice. This mapping of instances to trie nodes is different from $\mathbb{D}\langle X\rangle$, where Δ directly maps each θ to a monitor (state). In fact, $\mathbb{E}\langle X\rangle$ cannot map instances to monitors: it does not instantiate monitors at runtime and only instantiates monitors for unique traces in `trie` just before the program terminates. Also, unlike TraceMOP, LazyMOP does not use a separate map $\mathcal{B}$ from monitors to slices. Instead, LazyMOP directly retrieves slices from $\mathcal{D}$, reducing event processing time by avoiding an additional layer of map lookup.

Algorithm 3 LazyMOP's Monitoring Algorithm, $\mathbb{E}\langle X\rangle$. Differences with $\mathbb{D}\langle X\rangle$ are shown in green.

```
Inputs: same as $\mathbb{D}\langle X\rangle$ ; Outputs: T : set of unique traces, $T_v \subseteq T$ : unique traces that violate $M$'s spec]
Globals: $\mathcal{D} : [[X \rightharpoonup V] \rightharpoonup$ trie node], $\mathcal{T} : [[X \rightharpoonup V] \rightharpoonup$ integer], $\mathcal{U} : [X \rightharpoonup V] \rightarrow \mathcal{P}_f([X \rightharpoonup V])$,
        disable: $[[X \rightharpoonup V] \rightharpoonup$ integer], timestamp : an integer, trie : a trie-like data structure, lock : a lock
Initialization: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta$, timestamp $\leftarrow 0$, $T \leftarrow \{\}$, $T_v \leftarrow \{\}$, trie $\leftarrow$ root trie node
        $\mathcal{D}(\theta).m \leftarrow 0$ for any $\theta$
 1: procedure main($e\langle\theta\rangle, l$)
 2:   if $e$ is termination signal then
 3:     for all slice in trie.slices do                                  ▷ only returns unique trace slices
 4:       $\tau \leftarrow$ decode(slice) ; $T \leftarrow T \cup \{\tau\}$ ; monitor $\leftarrow$ new(M) ; verdict $\leftarrow$ run($\tau$, monitor)
 5:       if verdict == violation then $T_v \leftarrow T_v \cup \{\tau\}$; return
 6:   if needsLock() then lock.acquire(); processEvent($e\langle\theta\rangle, l$); if needsUnLock() then lock.release()
 7: procedure processEvent($e\langle\theta\rangle, l$)
 8:   if $\mathcal{D}(\theta)$ undefined then
 9:     addSlices($e\langle\theta\rangle$)
10:     if $\mathcal{D}(\theta)$ undefined and $e$ is a creation event then defineNew($\theta$)
11:     disable($\theta$) $\leftarrow$ timestamp; timestamp $\leftarrow$ timestamp + 1
12:   for all $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ s.t. $\mathcal{D}(\theta')$ defined do $\mathcal{D}(\theta') \leftarrow \mathcal{D}(\theta')$.advance(encode($e, l$));
13: procedure addSlices($e\langle\theta\rangle$)
14:   for all $X_e \in \text{enable}_{\mathcal{G}}^{X}(e)$(in reversed topological order) do
15:     if Dom($\theta$) $\not\subseteq X_e$ then
16:       $\theta_m \leftarrow \theta'$ s.t. $\theta' \sqsubset \theta$ and Dom($\theta'$) = Dom($\theta$) $\cap X_e$
17:       for all $\theta'' \in \mathcal{U}(\theta_m) \cup \{\theta_m\}$ s.t. Dom($\theta''$) = $X_e$ do
18:         if $\mathcal{D}(\theta'')$ defined and $\mathcal{D}(\theta'' \sqcup \theta)$ undefined then defineTo($\theta'' \sqcup \theta, \theta''$)
19: procedure defineNew($\theta$)
20:   for all $\theta'' \sqsubset \theta$ do if $\mathcal{D}(\theta'')$ defined then return
21:   $\mathcal{D}(\theta) \leftarrow$ trie.root; $\mathcal{T}(\theta) \leftarrow$ timestamp; timestamp $\leftarrow$ timestamp + 1; $\mathcal{D}(\theta).m \leftarrow \mathcal{D}(\theta).m + 1$
22:   for all $\theta'' \sqsubset \theta$ do $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
23: procedure defineTo($\theta, \theta'$)
24:   for all $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ do if disable($\theta''$) > $\mathcal{T}(\theta')$ or $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$ then return
25:   $\mathcal{D}(\theta) \leftarrow \mathcal{D}(\theta')$; $\mathcal{T}(\theta) \leftarrow \mathcal{T}(\theta')$; for all $\theta'' \sqsubset \theta$ do $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$; $\mathcal{D}(\theta).m \leftarrow \mathcal{D}(\theta).m + 1$
```

The `main` procedure (lines 1–6) in Algorithm 3 is the entry point of $\mathbb{E}\langle X\rangle$. A special end event that is not related to any spec—*e.g.*, a JVM shutdown hook in Java—is a termination signal indicating that program execution is about to terminate. On line 6, $\mathbb{E}\langle X\rangle$ calls processEvent($e\langle\theta\rangle$) on each spec-related event e to perform trace slicing and append e to appropriate slices based on θ.

Lazy monitoring. Explicitly tracking unique traces without monitoring them against specs while a program runs, and only monitoring them just before program termination is a simple optimization for online RV of tests that is novel to LazyMOP. Specifically, unlike in $\mathbb{D}\langle X\rangle$.processEvent which TraceMOP inherits, $\mathbb{E}\langle X\rangle$.processEvent merely stores unique traces; it does not eagerly monitor events as they are signaled. Rather, monitoring of unique traces is lazily deferred until the end event arrives, when lines 2–5 are run. Line 4 does four things: it (i) decodes each unique trace slice in trie—for fast trace store accesses, LazyMOP encodes each event and its location as an integer on line 12—to obtain trace τ; (ii) adds τ to T, the set of unique traces to return; (iii) instantiates a new monitor m; and (iv) checks if m accepts τ. On line 5, if τ caused m to transition to a violating state, then τ is added to T_v, the set of violating traces to return. The decode procedure (not shown) reverses the effect of encode: it splits an integer into two parts, one representing the event's name and the other is a unique integer ID representing the event's location. The decode procedure also maps that ID back to a line of code in the monitored program.

On-demand Synchronization. The `if` statements before and after the call to $\mathbb{E}\langle X\rangle$.`processEvent` on line 6 show LazyMOP's on-demand synchronization, another simple optimization that is novel for explicit-trace RV. In TraceMOP, eager monitoring of signaled events requires lock acquisition and release on the trace store before and after processing each event, respectively (line 5 in Algorithm 2). But, on-demand synchronization means LazyMOP only starts acquiring and releasing locks on its trace store after seeing evidence that the program is multi-threaded (§3.1.2 explains how LazyMOP finds such evidence). After that, the `if` conditions on line 6 will subsequently always return `true`.

Event Encoding and Decoding. One reason TraceMOP does not scale well is that it represents event names and locations as strings in its trace store. Doing so makes trace store accesses slow because it takes $O(|\text{event name}| + |\text{location string}|)$ time to check if a trie node contains an event name and location. Since TraceMOP traverses its trie once per signaled event, this cost becomes quite significant in projects where RV signals millions, or even billions of events. LazyMOP introduces event encoding and decoding as simple optimizations to allow checking if a trie node contains an event in $O(1)$ time, thereby speeding up trace store accesses. On line 12, encode returns a unique integer per unique (`e`, `loc`) pair, where `e` is an event name (*e.g.*, `init` for BAOS) and `loc` is the line of code that signaled the event (*e.g.*, `loc` for line 4 in Fig 2 is a fresh integer, say 99, that maps to `cc.mallet.types.tests.TestSerializable:TestSerializable.java:4`). Dually, decode obtains (`e`, `loc`) pairs from an integer encoding. For example, decoding 1585 yields the event name 1 (from the last four bits, 0001), indicating an `init` event, and the location ID 99 (from the first 28 bits).

The encode procedure (not shown) first assigns to each unique location a unique integer ID, and then combines that ID with the event name. Internally, event names in each spec are mapped at instrumentation time to unique integers (*e.g.*, `init` for BAOS is 1, `write` is 2, etc.). Retrieving the full location (package name, class name, plus line number) per event at runtime is costly. LazyMOP amortizes this cost by retrieving the full location per event location only once and mapping each full location to a unique integer ID in a data structure (elided in Algorithm 3 for simplicity). So, obtaining the location ID of subsequent events from the same location is a fast lookup. Also, LazyMOP uses this ID-to-full-location mapping in decode to obtain the location of each event in trie.

The algorithmic simplicity of these three LazyMOP optimizations belies the complexity of their implementations. For example, efficient implementation of event encoding and decoding required us to modify complex code in the AspectJ compiler that LazyMOP (and JavaMOP and TraceMOP) use for instrumentation. §3.1.2 provides more details on our LazyMOP implementation.

Other ways $\mathbb{E}\langle X\rangle$ differs from $\mathbb{D}\langle X\rangle$ and TraceMOP. The `processEvent`, `addSlices`, `defineNew`, and `defineTo` procedures in $\mathbb{E}\langle X\rangle$ only differ from their counterparts in $\mathbb{D}\langle X\rangle$ in how they use and update $\mathcal{D}$ (a map from each parameter instance to the node representing the current end of its trace slice) instead of Δ (a map from each parameter instance to the state of the monitor checking its trace slice). We highlight these differences in green in Algorithm 3. On line 12 of `processEvent`, whereas $\mathbb{D}\langle X\rangle$ and TraceMOP's algorithm call the transition function σ of the monitor for θ' to check e, *i.e.*, $\sigma(\Delta(\theta'), e)$, LazyMOP's $\mathbb{E}\langle X\rangle$ advances the slice τ for θ' by the encoded event. To advance τ, procedure advance (not shown) either adds a new node to `trie` or updates `trie` to reflect that the slice for θ' is now $\tau \cdot e$. In either case, $\mathbb{E}\langle X\rangle$ increments the count of parameter instances whose slices is $\tau \cdot e$ by 1, and decrements the count of parameter instances whose slices is τ by 1. On the right side of Fig 4, each trie node is associated with a variable `m` that tracks the number of parameter instances whose slices end at that node. Just before the program terminates, LazyMOP uses these counts to determine the frequency of each unique trace. For example, `m = 2` if two instances have the same slice. Procedure addSlices in $\mathbb{E}\langle X\rangle$ is almost identical to createNewMonitorStates in $\mathbb{D}\langle X\rangle$; the former differs from the latter in its straightforward use of $\mathcal{D}$ instead of Δ.

As in $\mathbb{D}\langle X\rangle$ (Algorithm 1), $\mathbb{E}\langle X\rangle$'s defineNew is invoked when a new parameter instance θ is encountered for which no previous less informative instance exists that can be the basis of monitoring θ's slice. Differently from $\mathbb{D}\langle X\rangle$, line 15 in $\mathbb{E}\langle X\rangle$.defineNew maps θ to trie.root and increments $\mathcal{D}(\theta).m$ by 1, indicating that a new empty slice was created. However, if a previous less informative instance θ' already exists in $\mathrm{Dom}(\mathcal{B})$, then instead of cloning monitors in defineTo as $\mathbb{D}\langle X\rangle$ does, $\mathbb{E}\langle X\rangle$ copies the slice of θ', maps θ to that copied slice, and increments $\mathcal{D}(\theta).m$ by 1. That way, by the time $\mathbb{E}\langle X\rangle$.processEvent reaches line 12, the slice for θ is advanced by e in trie.

Two important differences between LazyMOP and TraceMOP are hard to show in Algorithm 3. First, to retrieve slices for parameter instances from $\mathcal{D}$, LazyMOP uses a different indexing tree that integrates closely with trie. That is, unlike TraceMOP, which uses an intermediate map $\mathcal{B}$ and keeps the indexing tree separate from trie, LazyMOP's indexing tree interacts directly with trie to speed up event processing. Each node in the indexing tree points directly to a node in trie. So, when a new event occurs, LazyMOP can quickly locate and update trie using only the new event's parameter instance. In contrast, TraceMOP first retrieves the monitor for that parameter instance and then uses that monitor to find and update trie. Second, whereas TraceMOP uses one trie to track unique traces for all specs, LazyMOP uses one trie per spec. Using a separate trie per spec has two benefits: it (i) reduces the size of each trie, speeding up trie operations, and (ii) allows LazyMOP to traverse the trie and monitor unique traces in parallel.

3.1.1 LazyMOP's Correctness. Comparing processEvent, createNewMonitorStates, defineNew, and defineTo in Algorithms 1 with counterparts in Algorithm 3 shows that LazyMOP's $\mathbb{E}\langle X\rangle$ uses the same event handling and trace slicing steps as $\mathbb{D}\langle X\rangle$. So, $\mathbb{D}\langle X\rangle$'s proofs of correctness, which only apply to these steps, also apply to $\mathbb{E}\langle X\rangle$. The correctness of trie advancement and lazy monitoring—the two ways $\mathbb{E}\langle X\rangle$ differs from $\mathbb{D}\langle X\rangle$—is straightforward. A proof by induction on the length of the path from trie.root to the node in trie that parameter instance θ points to before processing e shows the correctness of trie advancement. The correctness of lazy monitoring follows from the fact that we use well-known methods for checking if an automaton accepts a word.

3.1.2 Implementation. LazyMOP has a Code Generator and a Runtime Verifier. The Code Generator only runs once offline to produce the Runtime Verifier from a set of specs. The Runtime Verifier is a Java agent [94] that monitors executions against specs; it is (i) easily integrated into any Java project; and (ii) reusable across projects. To produce the agent, Code Generator uses AspectJ [73] to generate instrumentation code from spec files. Also, from each spec, Code Generator uses JavaParser [66] to generate a manager class that connects AspectJ code to LazyMOP and a monitor class.
Event encoding. At runtime, the agent first instruments the code under test (CUT) and tests based on the specs, then transforms the instrumented code to represent the full location of each instrumentation site (*e.g.*, line 6 in Fig 2) as a unique integer ID. To do so, LazyMOP fetches the location object per instrumented event from AspectJ, then encodes the event's name (*e.g.*, tobytearray in Fig 1) and full location (*e.g.*, cc.mallet.types.tests.TestSerializable:TestSerializable.java:6 for tobytearray in Fig 2).
Lazy monitoring. LazyMOP monitors unique traces in a shutdown hook, just before JVM shutdown. There, for each unique trace, LazyMOP decodes the integer IDs to obtain event names and locations. By using one trie per spec, LazyMOP knows the spec for each decoded trace. Next, LazyMOP creates a new monitor object per decoded trace and checks if it accepts the sequence of event names in that trace. Lastly, LazyMOP persists decoded traces and those that violate specs to disk.

On-demand synchronization. LazyMOP starts synchronizing trace store accesses only *after* finding that the CUT uses multiple threads. LazyMOP detects the start of threads by listening for method calls related to concurrency-related Java APIs that create threads, *e.g.*, `Thread`, `ExecutorService`, `ThreadPoolExecutor`. (Our artifact contains a full list of methods that LazyMOP listens for.)

3.2 LazyMOP^e

LazyMOP^e is an initial step towards the first of six applications (A1-A6 in §1) that LazyMOP can enable. LazyMOP aims to speed up explicit-trace RV on a program version by *monitoring* fewer identical traces, but LazyMOP^e aims to do so across multiple versions by *generating* fewer identical traces. LazyMOP^e uses traces and code changes to perform targeted disabling of monitoring during CI if all events in repeatedly monitored identical traces (Definition 1) are in a method.

LazyMOP^e is motivated by findings from a formative study that we perform in this paper to better understand how repeatedly monitored identical traces commonly manifest in real-world open-source projects. So, we first report on that study in §3.2.1 before describing LazyMOP^e's design in §3.2.2. We introduce this definition for ease of presentation in the rest of this paper:
Definition 4. An *idempotently monitored method, or IMM* is one from which at least one event in a repeatedly monitored identical trace is signaled. A *single-IMM identical trace* has events from only one IMM, *e.g.*, τ_1 and τ_2 in §2 with only BAOS events from `cloneViaSerialization` in Fig 2. A *multi-IMM identical trace* has events that are signaled from multiple IMMs.

3.2.1 *Formative Study.*
We do an in-depth analysis of traces and IMMs in 1,432 projects from [51] where JavaMOP and TraceMOP work. Our appendix has more study details (project selection and characteristics, analysis, results, etc.). To save space, here we only answer our six study questions:

1. **What proportion of monitored methods are IMMs?** On average per project, 76.0 of 181.7 monitored methods (with at least one signaled event) are IMMs. The max (min) monitored-method count is 3,942 (1). In 473 projects, the ratio of IMMs to monitored methods is 50% or more.
2. **What proportion of all traces have events in IMMs?** Summed across all 1,432 projects, 90.4% of all 7,424,704,750 (non-unique) traces have events in IMMs. Only 9.6% of those traces have no event in an IMM and over 50% of all traces in 1,245 projects have events in an IMM.
3. **How frequent are single- vs. multi-IMM identical traces?** Across all projects, the median ratio of single-IMM traces to all identical traces is 97.2%; the mean is 88.7%. 1,352 projects have over 50% single-IMM identical traces like τ_1 and τ_2 in §2; more than 95% of identical traces in 850 projects are single-IMM. So, we initially design LazyMOP^e to target single-IMM identical traces, which do not require inter-procedural analysis. Future work can tackle multi-IMM identical traces.
4. **How many specs are typically involved in IMMs?** 73.6% of IMMs involve multiple specs. The spec count per IMM is less than five in 66.4% of cases, *i.e.*, few specs are often involved. But, one IMM involves 12 specs. So, faster RV by exploiting IMMs requires handling multiple specs per IMM. But, a technique (*e.g.* [98]) that uses heavyweight per-spec static analysis to exploit IMMs is unlikely to scale well in CI: that costly analysis must be repeated per spec and per version.
5. **Where are IMMs located?** 69.7%, 6.3%, and 23.9% of IMMs are in 3rd-party libraries, the CUT, and tests, respectively. So, bytecode-level analysis is needed to exploit IMMs in libraries.
6. **How often do methods that produce single-IMM identical traces depend on inputs or outputs of other methods?** Across all projects, 95,107 out of 108,815 IMMs (87.4%) do not depend on the inputs or outputs of other methods. So, LazyMOP^e currently targets IMMs that do not depend on inputs or outputs of other methods. Future work can target the others.

3.2.2 *Design.*
The challenge in LazyMOP^e is how to know ahead of time if a yet-to-be-generated trace will be identical to a previously seen trace. LazyMOP^e addresses this challenge in the CI

Algorithm 4 LazyMOP^e's Algorithm

```
Inputs: $\mathbb{E}\langle X\rangle$'s inputs, $m$ : method in version $v_2$, $m'$ : $m$ in old version $v_1$, $\mathcal{S}$ : specs, $\mathsf{T}_{v_1}$ : $v_1$'s unique traces
Outputs: $\mathsf{T}$ : unique traces in $v_2$, $\mathsf{T}_\mathsf{V}$: $\subseteq$ traces, unique traces that violate $M$'s spec in $v_2$
1: procedure premain($m$, $m'$, $\mathcal{S}$, $T_{v_1}$) :                 ▷ invoked once at instrumentation time
2:   if $m \neq m'$ or $m$ is affected by the change then return            ▷ skip changed or affected methods
3:   for all $s$ in $\mathcal{S}$ do
4:     $\mathsf{T_m} \leftarrow \{\}$; for all $\tau \in T_{v_1}$ s.t. $\tau$ is for $s$ and $\forall \mathsf{e} \in \tau$, e.loc is in $m$ do $\mathsf{T_m} \leftarrow \mathsf{T_m} \cup \{\tau\}$
5:     $\mathsf{T_s} \leftarrow \{\}$; for all $\tau \in T_{v_1}$ s.t. $\tau$ is for $s$ and $\exists \mathsf{e} \in \tau$, e.loc is in $m$ do $\mathsf{T_s} \leftarrow \mathsf{T_s} \cup \{\tau\}$
6:     if $|\mathsf{T_m}| == |\mathsf{T_s}|$ and $|\mathsf{T_m}| == 1$ then $m \leftarrow$ transform(m, s)
7:     if cannotViolate($m, s, T_{v_1}$) then $m \leftarrow$ transform$'$(m, s)   ▷ compiler guarantees $m$ cannot violate $s$
8: procedure main($e\langle\theta\rangle, l$) :                      ▷ invoked everytime an event for $s$ is signaled at runtime
9:   $\mathbb{E}\langle X\rangle$.main($e\langle\theta\rangle, l$)
```

context by first obtaining relationships among traces and IMMs in version v_1. Then, in future version v_2, LazyMOP^e identifies all (m, s) pairs, where m is an IMM that is not affected by code changes, s is a spec, and all traces for s in m were identical and had only events from m in v_1. Next, LazyMOP^e transforms each m to generate its trace for s only once in v_2. If m changes to m' or if m is unchanged but is affected by the changes, LazyMOP^e should re-monitor all traces in un-transformed m' (or m) in v_2. We have so far only empirically evaluated LazyMOP^e's safety—ability to find all new violations after a code change. §3.2.3 discusses LazyMOP^e's correctness and safety.

Algorithm 4 shows how LazyMOP^e works. Lines 1–7 show `premain`, which runs once at instrumentation time, before monitoring starts; it aims to transform only IMMs that generate single-IMM repeatedly monitored identical traces to only generate one such trace at runtime. Line 2 finds IMMs that should not be transformed as those that changed ($m \neq m'$) or are affected by changes. Line 4 checks if each unique trace τ from m in v_1 was generated from s and contains only events from m. If so, τ is added to T_m. Line 5 adds τ to T_s if *any* event (instead of *all* on line 4) in τ is from m.

Next, LazyMOP^e uses T_m and T_s to decide if method m should be transformed. Since LazyMOP^e only aims to generate fewer single-IMM identical traces, it requires $|T_s| == |T_m|$, *i.e.*, both sets of traces for s only have events in method m. Also, if $|T_m| = 1$, then m produces only one repeatedly monitored identical trace. So, LazyMOP^e only transforms m if line 6 ensures that m only generates single-IMM repeatedly monitored identical traces for s. The goal of `transform` is to cause s to be monitored in m only once, during m's first invocation, and to skip monitoring s during all subsequent invocations of m. (§3.2.4 describes how we implement `transform`.) Even if the condition on line 6 is not met, LazyMOP^e can still reduce wasted monitoring by transforming m if all (possibly un-identical) traces that are generated from m are guaranteed by the compiler to not violate s.

The `cannotViolate` procedure on line 7, whose current implementation is discussed in §3.2.4, returns whether the compiler can guarantee that traces generated in some method m cannot violate spec s. For example, specs that monitor correct usage of `Iterator` cannot be violated if they are only used in code that the Java compiler generates while desugaring a `foreach` loop to a `while` loop. If `cannotViolate` returns `true`, LazyMOP^e uses `transform'` to remove all instrumentation related to spec s, such that method m does not produce any events from s. This `transform'` differs from `transform`, where s is still monitored in m during the first invocation. Our goal is to monitor the CUT and third-party library code, so LazyMOP^e does not monitor the standard Java library: the specs we use check whether developer written code follows correct usage protocols of JDK APIs, and monitoring the standard library would add unnecessary overhead with no benefit to developers.

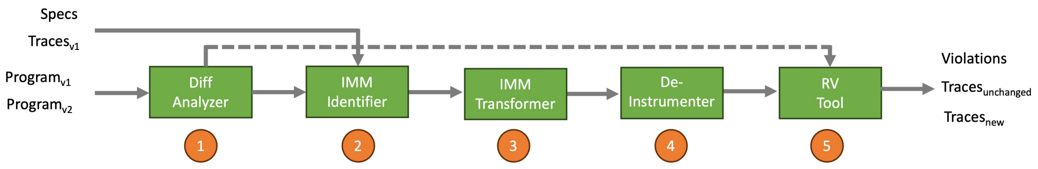

Fig. 5. The steps in LazyMOP^e's workflow.

3.2.3 *LazyMOP^e's Correctness and Safety.*

We discuss the correctness of Algorithm 4 theoretically in terms of its safety—ability to find all new violations after a code change—and trace preservation—ability to find all traces that LazyMOP finds after a code change.

Safety. Assuming T_{v_1} contains all unique traces in v_1 and tests are deterministic, only lines 2, 6, and 7 in Algorithm 4 can affect LazyMOP^e's ability to find all new violations that LazyMOP finds after code changes. So, we discuss why none of these lines make LazyMOP^e unsafe in theory.

1. If m changed or if it can behave differently because it transitively depends on input or output data from another method that changed, then line 2 is sufficient to ensure that LazyMOP^e does not apply any transformation and that it monitors m in the same way that LazyMOP does. In practice, this claim about line 2 assumes the availability of sound inter-procedural change-impact analysis and may not hold for multi-IMM identical traces, which we do not tackle in this paper.

2. Since line 6 ensures that all traces for s in m are single-IMM identical traces, `transform` preserves any violation found from the only trace of s generated from m after the transformation on line 6.

3. Since `cannotViolate` ensures that `transform`′ is only ever invoked if a compiler guarantees that m cannot violate s, the transformation on line 7 cannot lead to missing a violation.

Trace Preservation. We again discuss the impact of lines 2, 6, and 7 in Algorithm 4 on LazyMOP^e's ability to find all traces that LazyMOP finds. Trace preservation subsumes safety as a criterion for correctness: checking all unique traces ensures finding all violations, but not the other way around.

1. Line 2 ensures that if m changed or is affected by a change, then LazyMOP^e monitors m, and finds the same traces in m, as LazyMOP would.

2. On line 6, `transform` only changes the frequency of observing a single-IMM identical trace for s in m from $n \geq 1$ to 1. But, for the specs we check, changes in frequency do not affect bug finding.

3. On line 7, `transform'` causes LazyMOP^e to miss unique traces for s in m that LazyMOP finds. But, such missed traces cannot violate s and, for the specs we check, do not help find bugs.

3.2.4 *Implementation.*

Figure 5 shows the main steps in our LazyMOP^e implementation. Given specs, old version v_1, new version v_2, and unique traces from v_1, LazyMOP^e outputs violations, unchanged traces, and new traces. We next explain each LazyMOP^e step:

❶ Diff Analyzer compares the bytecode in v_1 and v_2 after "cleaning" all debug-related information so that classes where only, *e.g.*, white space and comments, changed are ignored. If class C's cleaned bytecode in v_2 differs from v_1's, or if C is new in v_2, C is marked as changed.

❷ IMM Indentifier finds the set of unchanged methods that produce single-IMM repeatedly monitored identical traces in v_2 using, specs, traces from v_1, and changed classes from ❶. Methods in changed classes that produced single-IMM identical traces in v_1 are fully re-monitored.

❸ IMM Tranformer transforms unchanged IMMs from ❷ before monitoring v_2. For each such IMM, all specs that do not produce single-IMM identical traces are first identified. Then, the bytecode of the class containing the IMM is transformed to add (i) a differently-named copy of the IMM; and (ii) code to the original IMM that dispatches second and subsequent calls of the original IMM to the copy. Next, the original IMM is instrumented with all related specs, but the copy is only partially instrumented with specs for which that IMM does not produce single-IMM identical traces. We describe an example of a transformed IMM in the Appendix.

❹ De-Instrumenter removes all instrumentation code from Java's `foreach` loops [65] for specs with only `Iterator` parameters, iff. those parameters are only used in the loop body. For example, consider the `Iterator_HasNext` spec: `iterator.next()` must be preceded by `iterator.hasNext()` on the same iterator, to avoid fetching non-existent elements. The Java compiler desugars such loops into `while` loops that use `Iterators` in ways that *cannot* violate such specs. So, de-instrumenting such loops will cause LazyMOP^e to miss unique traces, but it will not miss violations. Our artifact contains a list of all nine specs that De-Instrumenter supports.
❺ LazyMOP^e's agent (i) instruments transformed IMMs so that at runtime, after the first check, calls to the original IMMs are dispatched to the copies from ❸; (ii) monitors the instrumented code and reports any violations; and (iii) uses changed classes from ❶ to distinguish among traces from changed and unchanged classes; the former will be used as LazyMOP^e inputs in a future version.

We implement LazyMOP^e as a Maven plugin; it first runs between compilation and test execution to find changed classes using the same bytecode-level (and bytecode-cleaning) change detector used in several RTS tools [46, 47, 81, 83]. We implement IMM Tranformer in ASM [7]; it works on binaries and processes each class thrice: (i) map method names to method descriptors; (ii) copy IMMs and rename copies; and (iii) modify original IMMs to invoke copies after the first call.

Scope and Limitation. Fast and sound change-impact analysis would be needed if IMMs in unchanged classes become non-IMMs after changes because a *dependent* IMM depends on input or output of other methods. LazyMOP^e does not tackle dependent IMMs, but it can check for dependence by analyzing if the same traces result from multiple calls to an IMM. No independent IMM later became dependent in our evaluation. LazyMOP^e's De-Instrumenter only optimizes dependent IMMs if compiler-generated code cannot violate a spec.

4 Evaluation

Research Questions. We organize our LazyMOP and LazyMOP^e evaluation around five questions:

RQ1. How does LazyMOP compare with TraceMOP and JavaMOP in terms of time and memory overhead on *single* versions of open-source projects?

RQ2. How much does each new optimization in LazyMOP (lazy monitoring, event encoding, and on-demand synchronization) contribute to its efficiency?

RQ3. How do LazyMOP and LazyMOP^e compare with JavaMOP in terms of runtime overheads across *multiple* versions of evolving open-source projects?

RQ4. Can LazyMOP^e speed up SoTA evolution-aware RV techniques in eMOP [82, 131]?

RQ5. How safe is LazyMOP vs. TraceMOP, and how safe is LazyMOP^e vs. eMOP?

RQ1 compares LazyMOP's efficiency with those of explicit-trace TraceMOP and implicit-trace JavaMOP on single versions. RQ2 is an ablation study on the contribution of each optimization to LazyMOP's efficiency. RQ3 concerns LazyMOP^e's efficiency during evolution, vs. evolution-*unaware* LazyMOP and JavaMOP. RQ4 evaluates combinations of LazyMOP^e with two evolution-*aware* techniques in eMOP that aim to re-monitor only specs related to code affected by changes. Lastly, RQ5 checks (i) if LazyMOP preserves traces and violations that JavaMOP and TraceMOP find; and (ii) LazyMOP^e's and eMOP's safety, *i.e.* the ability to find new violations after a change.

Evaluation Subjects. We start our project selection by investigating if there are characteristics that indicate a project is a good candidate for evaluating the optimizations in LazyMOP and LazyMOP^e. To do so, we check for correlation between the ratio of TraceMOP time to LazyMOP time and various program characteristics, using all 1,432 projects from [51] where all tests still pass with JavaMOP. Like in [51], Table 1 shows that no strong correlation is observed for any characteristic, with or without outliers. There, the mutation score (*mut. score*) is computed using the 11 default operators in PIT [97], and coverage is computed using JaCoCo [64]. Notably, we find only a weak

Table 1. Pearson's correlation coefficients for LazyMOP time (s) over TraceMOP time (s) vs. several program characteristics: no. of test methods (#tests), test time w/o RV in seconds (t), relative JavaMOP overhead (t^{rv}/t), absolute JavaMOP overhead (s, $t^{rv}-t$), lines of code (SLOC), % statement coverage (cov^s), % branch coverage (cov^b), no. of GitHub commits (#SHAs), years since 1st commit (age), no. of stars (#★), max. total cyclomatic complexity (CyComp.), mutation score (mut. score), and ratio of trace repetitiveness (Rep.).

	#tests	t	t^{rv}/t	$t^{rv}-t$	SLOC	cov^s	cov^b	#SHAs	age	#★	CyComp.	mut. score	Rep.
Pearson's r (with outliers)	0.087	0.143	0.32	0.32	0.091	0.114	0.119	0.119	0.006	0.048	0.032	-0.009	0.231
Pearson's r (without outliers)	0.134	0.271	0.386	0.386	0.115	0.144	0.131	0.074	0.014	0.012	0.078	0.011	0.335

Table 2. Summary statistics on 179 projects that we evaluate. Table 1's caption describes column headers.

	#tests	t	t^{rv}/t	$t^{rv}-t$	SLOC	cov^s	cov^b	#SHAs	age	#★	CyComp.	mut. score	Rep.
Mean	349.6	24.5	12.3	209.3	13,424.3	64.2	57.2	482.8	10.7	298.5	155.7	76.2	97.8
Med	60	5.9	8.2	47.8	4,658	68.4	59.0	147	10	48	81.5	79.0	99.9
Min	1	1.5	1.1	3.2	93	0.1	0.0	3	3	0	6	21	41.3
Max	17,874	1,449.7	107.3	17,223.0	2.0×10^5	99.5	100.0	4,860	27	11,993	1,560	100	100.0
Sum	62,581	4,388.4	n/a	n/a	2.4×10^6	n/a	n/a	n/a	n/a	53,431	n/a	n/a	n/a

positive correlation with the characteristic *Rep.*, the amount of repeatedly monitored identical traces (*Rep.* = 100 * (*number of traces* − *number of unique traces*) / *number of traces*).

Given the lack of strong correlation, we split the 1,432 projects in Table 1 into three groups: (i) 1,233 projects where LazyMOP is less than five seconds faster than TraceMOP (912 projects) or TraceMOP is less than five seconds faster than LazyMOP (321 projects)—within this threshold, the choice of LazyMOP or TraceMOP is unlikely to matter in practice; (ii) 20 projects with average LazyMOP time of 55.7 seconds where TraceMOP is more than five seconds and at least 10% faster than LazyMOP, *i.e.*, LazyMOP's optimizations makes explicit-trace RV observably slower than TraceMOP for these few projects; and (iii) 179 projects where LazyMOP is more than five seconds and at least 10% faster than TraceMOP. Our evaluation uses the 179 projects in the third group, whose summary statistics are in Table 2; column headers are described in Table 1 and "n/a" are meaningless sums. (Our appendix summarizes the other two project groups.) The maximum value of Rep. shows up as 100.0% due to rounding; it is 99.99995%. The minimum test count is one; we keep that project since there is no strong correlation with #tests in Table 1. Mutation score (mut. score) is the ratio of killed to covered mutants. LazyMOP is not helpful when there are few repeatedly monitored identical traces. But trace repetitiveness (Rep.) is high for many of these projects.

Specs. We use all 84 specs in JavaMOP whose formalism (*e.g.*, LTL, ERE, FSM) compile into FSMs. Table 3 summarizes these 84 specs. There, "types" is the number of types of parameters per spec, "methods" is the number of methods each spec concerns, "formalism" is the logic used, and "#" is the number of specs in each formalism. 62, 16, 5, and 1 specs have one, two, three, and four parameter types, respectively. Of 84 specs, four concern only one method, 23 concern two methods, 23 concern three methods, 11 concern four methods, seven concern five methods, 15 concern more than five but fewer than ten methods, and one concerns 15 methods.

Table 3. Summary of the 84 specs.

	types	methods	formalism	#
Avg	1.3	3.7	ERE	69
Med	1	3	FSM	13
Min	1	1	LTL	2
Max	4	15		
Sum	113	310		84

Baselines. We compare the proposed techniques with JavaMOP, TraceMOP, and two techniques from eMOP: ps_1^c and $ps_3^{c\ell}$. §2 described JavaMOP and TraceMOP. We use the JavaMOP implementation in TraceMOP [125], which recently updated and refactored JavaMOP to support many modern Java features [53]. We choose JavaMOP because it was used in all prior work on RV of tests and it is the only RV tool that was shown to scale for simultaneously monitoring scores of the kind of specs that we use in thousands of open-source projects [51, 52, 67, 79, 82, 92]. Also, TraceMOP is the only online explicit-trace MOP-style RV technique today. We compare LazyMOP and LazyMOP^e with eMOP, the only evolution-aware MOP technique and tool today that targets

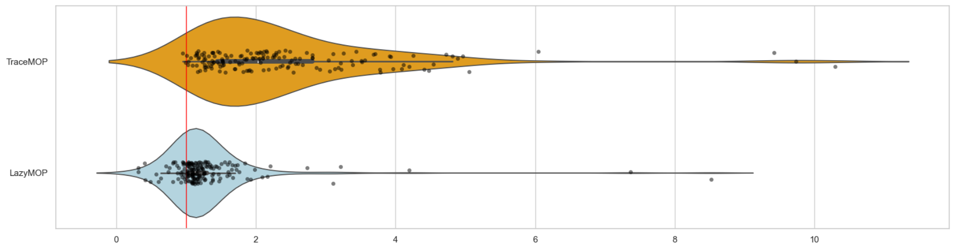

Fig. 7. Ratio of TraceMOP time (top, for 161 projects) and LazyMOP time (bottom, for 179 projects) to JavaMOP time (the red line).

the reduction of monitoring costs. iMOP [52] is also evolution-aware, but it targets reduction of instrumentation costs [52], which is complementary but out of scope in this paper.

We use two baselines from eMOP: $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$; they are, respectively, the fastest safe-by-design and fastest unsafe-by-design techniques among the 12 that speed up RV by re-monitoring only a subset of specs affected by code changes. Safety means finding all violations that are new after a change, assuming deterministic tests and within static analysis limitations [82]. $\boldsymbol{ps}_1^{c}$ uses a conservative change-impact analysis and re-monitors specs with events in (i) changed classes δ; (ii) dependents of δ; (iii) *dependees* of δ, *i.e.*, unchanged classes that are not in (i) and (ii) but which can generate new events in the new version if δ now passes new data to them; and (iv) dependees of dependents of δ. These four sets can be a large portion of projects' classes, so $\boldsymbol{ps}_1^{c}$ tends to re-monitor a large fraction of specs and it is often slower than $\boldsymbol{ps}_3^{c\ell}$, which is designed to trade safety for efficiency. $\boldsymbol{ps}_3^{c\ell}$ uses a less conservative analysis and only re-monitors specs with events in δ and dependents of δ. $\boldsymbol{ps}_3^{c\ell}$ also does not monitor third-party libraries.

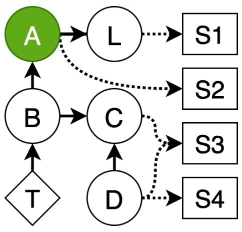

Fig. 6. eMOP example.

Fig 6 shows how $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ work. There, circles A, B, C, and D are CUT classes; L is a library class; and T is a test class. Rectangles S1–S4 are specs. Solid arrows from class x to class y means x extends or uses y. Dashed arrows from class x to spec Z means x may generate Z events. Suppose only A changed in the new version (green in Fig 6), $\boldsymbol{ps}_1^{c}$ finds classes A, B, C, L, and T as impacted. So, $\boldsymbol{ps}_1^{c}$ re-monitors only S1–S3; under the stated assumptions, new S4 events cannot be generated in these impacted classes. But, $\boldsymbol{ps}_3^{c\ell}$ finds only A, B, and T as impacted: L is a library class and a dependee of δ, and C is a dependee of a dependent of δ. So, $\boldsymbol{ps}_3^{c\ell}$ re-monitors only S2 in the new version. We use the $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ implementations in Yorihiro et al.'s eMOP Maven plugin [131].

Experimental Setup. We write scripts to automate our experiments, *all of which use RV to only monitor passing tests.* We modify the Maven extension from [51] to integrate JavaMOP, TraceMOP, LazyMOP, and LazyMOP^e into projects. Answering RQ2 requires profiling RV runs, so we use a Maven extension to integrate the `async-profiler` [8] into evaluated projects. We use a bash script to monitor peak program memory usage. We perform all experiments in Docker or Singularity containers; the corresponding Docker and Singularity files and usage instructions are in our artifacts. We run experiments on two machines, depending on projects' memory requirements, but we run all experiments for each project on the same machine: (i) an Intel® Xeon® w9-3475X (36 cores, 72 threads) CPU, 128 GB RAM, Ubuntu 24.04, Java 8, and Maven 3.8.8; (ii) Intel Core i7-13700k (16 cores, 24 threads) CPU, 64 GB RAM, Ubuntu 22.04, Java 8, and Maven 3.8.8.

4.1 RQ1: LazyMOP vs. JavaMOP, TraceMOP on One Program Version

We compare LazyMOP's time and memory overheads vs. TraceMOP and JavaMOP on 179 projects, one version each. The goal is to evaluate LazyMOP's performance against SoTA explicit-trace RV (*i.e.*, TraceMOP) and implicit-trace RV (*i.e.*, JavaMOP).

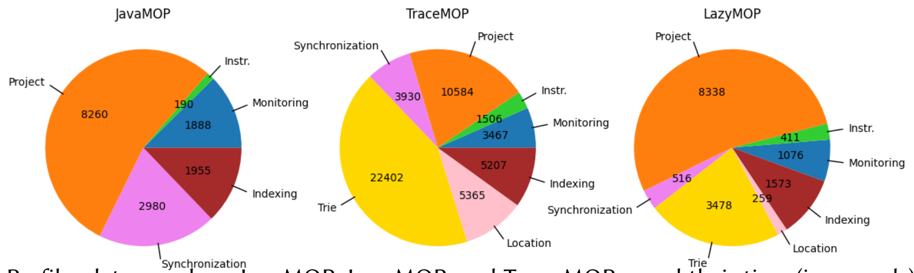

Fig. 8. Profiler data on where JavaMOP, LazyMOP, and TraceMOP spend their time (in seconds): in the project (Project), instrumentation (Instr.), monitoring (Monitoring), accessing indexing tree (Indexing), fetching locations (Location), accessing `trie` (Trie), and managing locks (Synchronization).

Runtime Overheads. The x-axis in Fig 7 shows the ratios of time for TraceMOP over JavaMOP (top) and LazyMOP over JavaMOP (bottom); 1 represents JavaMOP time. We highlight two main results. First, TraceMOP crashes with out-of-memory errors in 18, or 10.1% of projects. Of these 18 projects, LazyMOP is faster than JavaMOP in four and is only 12.6% slower on average (max: 25.9%) than JavaMOP in the other 14. Second, excluding projects where TraceMOP crashes, LazyMOP is faster than TraceMOP in all 161 projects by up to 4.9x or 6.5 hours (average: 1.9x or 4.1 minutes).

Surprisingly, LazyMOP is faster than JavaMOP in 33, *i.e.*, 18.4% of, projects, by an average of 1.4x (max: 3.2x). We find that these 33 projects have many (637,769,089 out of 637,927,704, or 99.98%) repeatedly monitored identical traces (so, lazy monitoring pays off), and are single threaded (so, on-demand synchronization pays off). Also, LazyMOP is only less than 10% slower than JavaMOP in another 102, *i.e.*, 57.0% of projects. In all 179 projects, LazyMOP is only 1.3x slower than JavaMOP (as expected: LazyMOP tracks traces but JavaMOP does not) on average (max: 8.5x).

Since LazyMOP is (i) faster than explicit-trace TraceMOP in all projects; (ii) faster than implicit-trace JavaMOP in 18.4% of projects; and (iii) only less than 10% slower than JavaMOP in another 57.0% of projects, we conclude that LazyMOP is faster than TraceMOP for RV of tests. So, we do not evaluate TraceMOP in RQ3–RQ5.

Memory Overheads. LazyMOP's peak memory is 1.6x less than TraceMOP's on average (max: 4.8x), so LazyMOP's event encoding pays off. But, LazyMOP's peak memory is higher than TraceMOP's on 28 projects where memory needed for maintaining a trie-like structure per spec outweighs the benefits of event encoding. On the other hand, LazyMOP's peak memory across all projects is 2.7x higher than JavaMOP's on average (max: 19.7x): LazyMOP stores unique traces and tracks many parameter instances (avg: 10,588,301, sum: 1,895,305,876). We conclude from these peak memory comparisons that LazyMOP is also often more space efficient than TraceMOP.

4.2 RQ2: Contributions of LazyMOP Components to its Efficiency

Figure 8 shows profiler data from our ablation study on how much lazy monitoring, event encoding, and on-demand synchronization contribute to LazyMOP's efficiency. There, we see proportions of time that LazyMOP, TraceMOP, and JavaMOP spend in their various components, aggregated across 161 projects (caption explains colors).

We highlight three main findings. First, the "Monitoring" portions show that LazyMOP's lazy monitoring incurs much less time than eager monitoring in JavaMOP and TraceMOP. Second, explicit-trace LazyMOP and TraceMOP store unique traces in trie-like data structures, but LazyMOP's time to do so ("Trie") is much faster than TraceMOP's, due to event encoding and LazyMOP's seamless integration of its indexing tree and trie-like structure. That integration also makes LazyMOP spend less time than TraceMOP on accessing indexing trees ("Indexing"). Lastly, LazyMOP spends less time managing locks ("Synchronization") than JavaMOP and TraceMOP; a benefit of on-demand synchronization.

The overall time differences among LazyMOP, TraceMOP, and JavaMOP should be interpreted in terms of average per-event processing time. For example, in project `jingpeicomp/id-generator`, where TraceMOP crashes, LazyMOP is around 30 seconds faster than JavaMOP because LazyMOP's average per-event time of 37 nanoseconds (obtained by dividing the sum of times in `id-generator`'s profiler data—104.84 seconds—by its number of events, 2.8 billion) is lower than JavaMOP's 48 nanoseconds. That difference accumulates across all 2.8 billion events in `id-generator` to over 30 seconds, but it is not only due to lazy vs. eager monitoring. LazyMOP and JavaMOP each spend 65 seconds to store events in indexing trees ("Indexing"). LazyMOP spends only nine seconds to transition monitors ("Monitoring") at the end for only 352 unique traces, while JavaMOP spends 27 seconds because it eagerly repeatedly and identically monitors these 352 unique traces 97 million times. Separately, JavaMOP spends 23 seconds to manage locks ("Sync"), compared to LazyMOP's five seconds. But, LazyMOP spends nine seconds in its trie-like structure ("Trie") and three seconds to look up locations ("Location")—costs that JavaMOP does not incur. Results in projects where LazyMOP is slower than JavaMOP can also be explained in terms of similar analysis of average time per event.

To see the contributions of event encoding and on-demand synchronization, which can be turned on and off, we measure LazyMOP's overhead without these optimizations. (Lazy monitoring cannot be turned on/off.) Table 4 shows LazyMOP times with all optimizations (t^{lazy}), with only lazy monitoring and on-demand synchronization (NoIntEncode), with only lazy monitoring and event encoding (NoOnDemand), and with only lazy monitoring (NoBothTime). The results in Table 4 show that all three optimizations in LazyMOP contribute to its efficiency. For all summary statistics, the time for full LazyMOP (t^{lazy}) is less than the times for all other columns. Lazy monitoring alone (NoBothTime) is slower than other configurations, but LazyMOP still takes less time (44,885.6s) than TraceMOP (62,388.1s). Lastly, event encoding alone (NoIntEncode) contributes more to LazyMOP's efficiency than on-demand synchronization alone (NoOnDemand). Overall, we conclude that event encoding and on-demand synchronization are essential to LazyMOP's efficiency, but event encoding contributes more.

Table 4. Times (s) with and w/o LazyMOP optimizations.

	t^{lazy}	NoIntEncode	NoOnDemand	NoBothTime
Mean	216.8	333.3	234.0	355.7
Med	71.9	79.0	76.0	81.3
Min	6.5	7.4	8.2	8.5
Max	11,267.8	22,722.3	11,362.1	24,894.0
Sum	38,807.8	59,666.0	41,889.8	63,678.5

4.3 RQ3: LazyMOP and LazyMOP^e vs. JavaMOP as Software Evolves

Here and in RQ4 (§4.4), we use 2,401 versions in 104 of 179 projects where we find at least four versions where Java file(s) changed, code compiles, and tests pass with and without JavaMOP and LazyMOP. The other 47 projects have fewer than four versions where JavaMOP works, 21 have versions with compilation or dependency errors, and seven have versions with failing tests.

The first six columns in Table 5 show how many times JavaMOP, LazyMOP, and LazyMOP^e perform best in terms of end-to-end time. Our appendix has detailed comparisons. LazyMOP^e is faster than JavaMOP in 38 (36.5% of) projects. But, LazyMOP is only faster than JavaMOP in 17 cases, *i.e.* in 16.3% of these projects. Comparing LazyMOP with LazyMOP^e in Table 5, LazyMOP^e is faster than LazyMOP in 53 cases, *i.e.*, in 51.0% of these projects, by an average of 1.5x (max: 3.9x). So, LazyMOP^e's optimizations often make it faster than LazyMOP and JavaMOP during evolution.

Fig 9 shows how much the end-to-end times of LazyMOP (left) and LazyMOP^e (right) are faster (blue bars) or slower (red bars) than JavaMOP in 104 projects. Our analysis of 66 projects where LazyMOP^e is slower than JavaMOP shows that in 55 (83.3%) of them, over 50% of RV time is spent

Table 5. Pairwise comparisons of techniques during evolution. Parentheses show how much (avg/max) faster a technique is. $\boldsymbol{ps}_1^{c+imm}$ combines $\boldsymbol{ps}_1^{c}$ with LazyMOP^e, and $\boldsymbol{ps}_3^{c\ell+imm}$ combines $\boldsymbol{ps}_3^{c\ell}$ with LazyMOP^e.

JavaMOP	LazyMOP	JavaMOP	LazyMOP^e	LazyMOP	LazyMOP^e	$\boldsymbol{ps}_1^{c}$	$\boldsymbol{ps}_1^{c+imm}$	$\boldsymbol{ps}_3^{c\ell}$	$\boldsymbol{ps}_3^{c\ell+imm}$
87 (1.4/8.4x)	17 (1.3/2.9x)	66 (1.5/8.8x)	38 (1.6/3.5x)	51 (1.1/1.8x)	53 (1.5/3.9x)	61 (1.4/7.7x)	43 (1.5/3.6x)	74 (1.3/8.0x)	30 (1.5/4.6x)

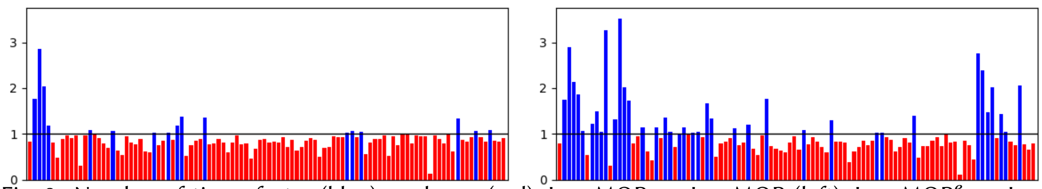
Fig. 9. Number of times faster (blue) or slower (red): LazyMOP vs. JavaMOP (left); LazyMOP^e vs. JavaMOP (right).

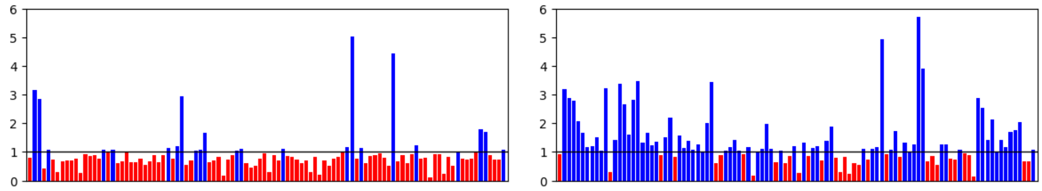
Fig. 10. Similar to Fig 9, but with instrumentation cost excluded.

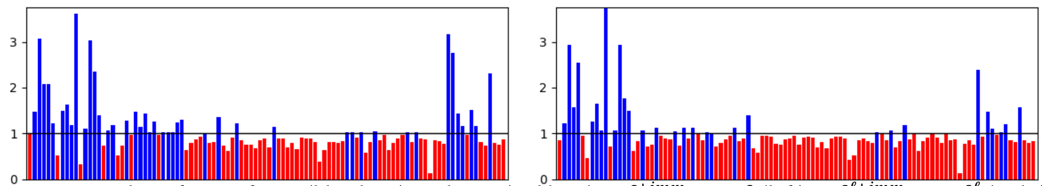
Fig. 11. Number of times faster (blue bars) or slower (red bars): $\boldsymbol{ps}_1^{c+imm}$ vs. $\boldsymbol{ps}_1^{c}$ (left); $\boldsymbol{ps}_3^{c\ell+imm}$ vs. $\boldsymbol{ps}_3^{c\ell}$ (right).

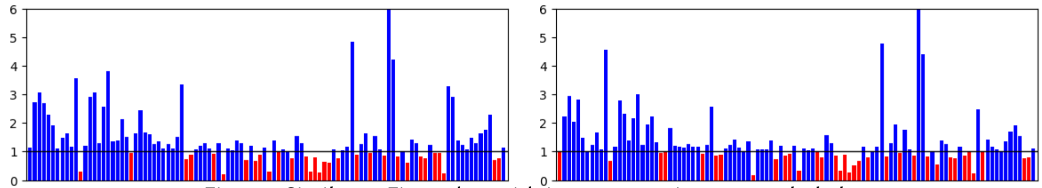
Fig. 12. Similar to Fig 11, but with instrumentation cost excluded.

on instrumentation. Fig 10 shows the speedups when instrumentation time is excluded. There, we see that LazyMOP^e is faster than JavaMOP in 71 (68.3%) projects and slower in 33 projects.

Overall, we conclude that LazyMOP^e is faster than LazyMOP as software evolves, but more work is needed to make LazyMOP^e faster by (i) combining instrumentation-driven techniques like iMOP [52] with LazyMOP^e; and (ii) handling more IMMs. Some major IMM categories that future work should target are IMMs that depend on input or output of other methods and IMMs with repeatedly monitored identical traces with events from multiple methods.

4.4 RQ4: Combining LazyMOP^e with Existing Evolution-aware RV Techniques

Given the promise shown by LazyMOP^e in RQ3, a natural question is whether it can be combined with eMOP techniques to make them faster. To answer that question, we combine $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ (which use JavaMOP to re-monitor a subset of specs after code changes) with LazyMOP^e (which generates and monitors fewer identical traces than JavaMOP). Then we compare our combinations with the $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ implementations in Yorihiro et al.'s eMOP tool [131].

The seventh and eighth columns in Table 5 compare our combination of LazyMOP^e with $\boldsymbol{ps}_1^{\mathrm{c}}$, named $\boldsymbol{ps}_1^{\mathrm{c}+imm}$, with $\boldsymbol{ps}_1^{\mathrm{c}}$ alone, using end-to-end times, on the same projects and versions in RQ3. $\boldsymbol{ps}_1^{\mathrm{c}+imm}$ is faster than $\boldsymbol{ps}_1^{\mathrm{c}}$ in 43, *i.e.*, 41.3% of projects by 1.5x on average (max: 3.6x). For the other 61 projects, $\boldsymbol{ps}_1^{\mathrm{c}+imm}$ is slower by 1.4x on average (max: 7.7x). Similarly, the ninth and tenth columns in Table 5 compare our combination of LazyMOP^e with $\boldsymbol{ps}_3^{\mathrm{c}\ell}$, named $\boldsymbol{ps}_3^{\mathrm{c}\ell+imm}$, with $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ alone. LazyMOP^e-based $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ is faster than $\boldsymbol{ps}_3^{\mathrm{c}\ell+imm}$ in 30, *i.e.*, 28.8% of projects by 1.5x on average (max: 4.6x). For the other 74 projects, $\boldsymbol{ps}_3^{\mathrm{c}\ell+imm}$ is slower by 1.3x on average (max: 8.0x).

Fig 11 again shows the speedup (blue bars) and slowdowns (red bars) of $\boldsymbol{ps}_1^{\mathrm{c}+imm}$ vs. $\boldsymbol{ps}_1^{\mathrm{c}}$ (left) and of $\boldsymbol{ps}_3^{\mathrm{c}\ell+imm}$ vs. $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ (right) in all 104 projects, using end-to-end times. In 52 of 61 projects (85.2%) and 59 of 74 projects (79.7%), where $\boldsymbol{ps}_1^{\mathrm{c}}$ or $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ performs better, RV overhead is dominated by instrumentation. Fig 12 shows a similar plot after excluding instrumentation time. There, $\boldsymbol{ps}_1^{\mathrm{c}+imm}$ is faster than $\boldsymbol{ps}_1^{\mathrm{c}}$ in 73 (70.2%) projects, and $\boldsymbol{ps}_3^{\mathrm{c}\ell+imm}$ is faster than $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ in 70 (67.3%) projects. Overall, we conclude that combining with LazyMOP^e speeds up these existing evolution-aware RV techniques in several projects.

4.5 RQ5: Violation Preservation, Trace Preservation, and Safety

Violation Preservation. We check if LazyMOP finds the same violations as JavaMOP and TraceMOP in single-version experiments (RQ1). To do so, we compare the sets of unique violations reported by these techniques. (A violation can occur multiple times if a violating trace is in a loop or in a method that is called multiple times.) JavaMOP's implicit-trace RV makes it hard to tell if two violations have the same trace, so we follow prior work [79]: two violations of a spec are the same if the code location of the last event in their violating trace is the same. In all 179 projects, JavaMOP and LazyMOP find 2,099 violations in single-version experiments. But, TraceMOP only finds 1,736 violations; all 363 missed violations are in the 18 projects where TraceMOP crashed.

Also, we randomly sample 100 violations of specs that helped find bugs in prior work [77, 79, 92]. We find 21 violations (from four specs and 11 projects) that are bugs. Of these, 13 are violations of the `BAOS` spec in §2.1. The other 79 violations that we inspect are false alarms. These rates are on par with prior work [77, 79, 92], which find that false alarms are mostly due to imprecision in specs. We are in the process of reporting these bugs to the developers of these projects.

Unique Trace Preservation. We measure how many traces monitored by TraceMOP are also monitored by LazyMOP. This evaluation is important because trace preservation is a stronger quality criterion for RV than violation preservation—the former guarantees the latter, but not vice versa. In this measurement, we account for non-determinism by taking the union per technique across four runs on the same machine. Figure 13 shows a Venn diagram comparing the sets of violations from each tool. 1,308,609 unique traces found by both LazyMOP and TraceMOP are identical. LazyMOP also finds 151,788 unique traces that TraceMOP does not, while TraceMOP finds 151,802 traces that LazyMOP does not.

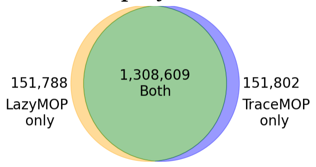

Fig. 13. Venn diagram for no. of unique traces found by LazyMOP only, TraceMOP only, and both.

On average across these projects, 89% of the union of all traces are found by both techniques. Since all tests pass in our evaluation, we have no evidence that they are due to the widely studied *flaky tests*, which sometimes pass and sometimes fail on the same code [2, 15, 54, 59, 75, 86, 95, 100, 112, 115, 118]. So, we hypothesize that sources of non-determinism that did not affect test outcomes are responsible for these differences. To test our hypothesis, we compare a subset of 100 traces from the other 11%—a task made possible by the fact that LazyMOP and TraceMOP collect traces. The sources of test non-determinism that we find are: (i) relying on random number generation (*e.g.*, `Random.nextInt()`), (ii) random shuffling of collections (*e.g.*, `Collections.shuffle()`), (iii) relying on the system environment (*e.g.*, environment variables or system properties that change across our runs of different techniques on the same machine), and (iv) a combination of (i) and (ii). Future work should investigate how to make RV more reliable in the presence of non-deterministic tests. Since we witnessed executions where each trace occurs, tests always pass in our experiments, and we find test non-determinism caused differences among monitored traces, we conclude that LazyMOP finds the same set of traces as TraceMOP, modulo test non-determinism.

Safety. Finally, we check if LazyMOP^e finds the same violations as JavaMOP in multi-version experiments (RQ3 and RQ4) and the safety of $\boldsymbol{ps}_1^{c+imm}$ of $\boldsymbol{ps}_3^{c\ell+imm}$. (Evolution-aware RV assumes developers are aware of old violations, so they can miss violations that were in the old version.) LazyMOP^e finds the same violations as JavaMOP. That is, LazyMOP^e is safe in our experiments. In all 2,401 versions in these projects, JavaMOP finds 1,152 new violations. But, $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_1^{c+imm}$ miss four violations due to a known bug in eMOP [22]. Also, the 10 violations missed by $\boldsymbol{ps}_3^{c\ell}$ and $\boldsymbol{ps}_3^{c\ell+imm}$ are expected, and are due to these techniques trading safety for performance. We conclude that combining LazyMOP^e with $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ does not make them less safe in our experiments.

5 Discussion

5.1 Future Work, Threats to Validity, and Limitations

5.1.1 Future Work. Other IMM Categories. In the future, we plan to investigate how to exploit multi-IMM identical traces in LazyMOP and LazyMOP^e. Doing so will require fast and sound inter-procedural analysis to determine when it is safe to skip the generation or monitoring of such traces. We also plan to investigate whether repeated monitoring *within* traces can be reduced. LazyMOP reduces repeated monitoring *across* identical traces (*i.e.*, when an identical trace is repeatedly checked against the same spec). Our manual inspection shows that reducing repeated monitoring *within* traces could further speed up LazyMOP and LazyMOP^e. We find that identical sub-traces often repeat and all events in such sub-traces are from one method. So, it may be possible to de-instrument methods related to such sub-traces after monitoring them the first time.

Improving LazyMOP^e. We only exploit one pattern (*i.e.*, for-each loops) when evidence exists in the old version's traces or from the compiler that there can be no violation in the new version. We plan to find and exploit more of such patterns. We only empirically evaluate LazyMOP^e's correctness/safety in this paper. We plan to investigate the theoretical correctness of LazyMOP^e, and how to make it safe if unchanged IMMs can generate new events in the new version because of modifications to changed code. Lastly, we plan to find ways to make LazyMOP^e faster in more projects by reducing its instrumentation-related costs in instrumentation-dominated projects.

Improving LazyMOP's time and space efficiency. LazyMOP shows promising results in terms of how much faster it is and how much less peak memory it uses compared to the SoTA explicit-trace RV technique, TraceMOP. But, there is still plenty of room to speed up LazyMOP and optimize its memory consumption. Doing so as part of future work, *e.g.*, by investigating other data structures, can make LazyMOP (and therefore LazyMOP^e) to be more efficient.

5.1.2 Threats to Validity. Our results may not generalize beyond the 179 projects that we evaluate. To mitigate this threat, before selecting evaluation subjects, we first show that LazyMOP works on 1,432 projects from [51], and that it is faster than or about the same as TraceMOP on single versions of 98.6% of those projects. We might have inadvertently introduced errors in our implementation. To mitigate that threat, we validate our tools by checking whether they produce the same violations as JavaMOP, which has been widely used. Non-deterministic tests make RV tools in this paper observe different traces in different runs on the same machine. To mitigate this threat, we run experiments multiple times and take the union of violations and traces across runs. We also identify some sources of non-deterministic executions that make traces differ across runs; future work can build on these findings. The main threat to *external validity* consists of the selection of evaluation projects. So, we use popular projects from prior work and define exclusion criteria (§4).

5.1.3 Limitations. This paper focuses on reducing RV overhead during testing. Other problems that impede RV usage in practice are out of scope. Such problems include improving spec quality, making spec languages more user friendly, automatically inferring better-quality specs, reducing the time it takes to check if spec violations are true bugs (or due to imprecision in specs), etc. These problems are being investigated elsewhere [45, 50, 77, 79, 92, 104, 124], and they are orthogonal to the one addressed in this paper. Also, the work in this paper is limited to the monitoring-oriented programming (MOP) RV style [25, 29, 91]. Other important RV styles exist [10–12, 14, 20, 34–37, 44, 61, 63, 90, 101, 103, 127], and future work should investigate if the ideas in LazyMOP and LazyMOP^e can speed them up. Lastly, our work may be limited to the kinds of API-level specs that we use. But, those specs are part of the largest publicly-available set.

6 Related Work

Repetitive Monitoring and Testing. Purandare et al. [98] observed that RV overhead when monitoring one spec against the DaCapo benchmarks was often due to repeated monitoring within loops. So, they proposed a static analysis to find when a loop stutters—intuitively, the iteration after which no new trace will be observed. They also develop a framework to transform loops so that monitoring only occurs before stuttering. Like them, we are also interested to optimize RV by reducing repeated and wasteful monitoring. But, their work is not concerned with the notions of lazy monitoring, event encoding, and on-demand synchronization that underlie LazyMOP. Also, LazyMOP^e is complementary, allowing us to more simply solve a more general problem at a higher granularity level without needing heavy-weight static analysis per loop and spec pair (more on that below). We are also the first to conduct an empirical study of IMMs and tackle the problem of repeatedly monitoring identical traces during software evolution.

LazyMOP^e is simpler than Purandare et al.'s approach: LazyMOP^e requires only the traces from a prior RV run. We solve a more general problem because (i) repetitive monitoring during testing can happen if multiple tests cover a program path, even when no loops are involved; and (ii) multiple specs can be involved in an IMM (their approach handles only one spec at once per loop). LazyMOP^e handles both cases, but Purandare et al.'s approach cannot handle the first, and it must perform additional analysis in the second. Also, unlike Purandare et al.'s approach, LazyMOP^e has no limitations when it comes to exceptional program paths. We report end-to-end times, but their static analysis times are not included in their paper (but they report that it can take hours). However, we think that future work should revisit Purandare et al.'s approach during testing, *e.g.*, by leveraging information about code changes during software evolution to reduce its costs.

Instrumentation-based Optimization. To our knowledge, LazyMOP^e is the first to apply de-instrumentation to optimize RV of tests. But, other researchers looked at speeding up RV using instrumentation-based optimization. For example, several works use static analysis to selectively instrument only parts of a program that static analysis could not prove safe [19, 38, 122]. Other researchers speed up RV by reducing instrumentation costs, *e.g.*, Bodden et al. distribute instrumentation costs across users [18] and Navabpour et al. develop instrumentation for sampling events [93]. These works are not concerned with RV during testing or with de-instrumentation. Other dynamic analyses used de-instrumentation for optimization, *e.g.*, a dynamic data-race detection (LiteRace [89]) and a memory-leak detector (SWAT [60]) first create instrumented and un-instrumented copies of functions. Then, at runtime, they adaptively sample calls to either copy. LazyMOP^e also creates a copy of IMMs, but does not instrument the related spec in the copy or use (adaptive) sampling. In CBI [6], instead of de-instrumenting after the fact, like LazyMOP^e does, the instrumentation process itself samples what parts of the program to instrument.

Offline RV. Offline RV [13, 33, 71, 84] is an explicit-trace RV approach, but monitoring is inactive as a program runs. A program trace (without any slicing) is collected as the program runs, and then processed in a separate offline phase. In a sense, LazyMOP is a hybrid online-and-offline approach that collects and slices unique traces at runtime and then monitors them *just before* the program terminates. It is not clear if purely offline RV can fit in tight CI budgets.

RV during Software Testing. Early work combined RV with automated test generation [4, 5] and allowed developers to write and monitor specs within unit-test frameworks [37, 103]. More recent works (i) demonstrated that RV during testing helps find many bugs [77, 79]; (ii) developed evolution-aware RV [80, 82, 131]; and (iii) used machine learning to reduce human time to inspect violations [92]. LazyMOP^e is motivated by this line of work on RV during regression testing in general, and the recent study of RV overheads during testing in particular [51].

Regression Test Selection. Evolution-aware RV [80] was inspired by regression test selection (RTS) [130], which aims to speed up regression testing by re-running only tests affected by code changes. Many RTS techniques [21, 58, 106, 108, 121, 126, 128, 136] and studies [9, 17, 39, 40, 49, 55, 78, 107, 109, 116, 117, 119, 120] exist, and recent work started getting adopted in industry [46, 47, 88]. RTS techniques can be categorized based on whether they use static [72, 81, 114, 129], dynamic [46–48], hybrid [83, 132, 134], or predictive [85, 88, 133] change-impact analysis to identify affected tests. eMOP and our combinations of eMOP with LazyMOP^e use static analyses. So, other kinds of analysis can be investigated in the future. But, in our experience, using dynamic analysis in evolution-aware RV requires running tests twice—once to find relationships of code with specs, and once to re-monitor affected tests—an overhead of at least $2x$ vs. running tests without RV.

RTS differs from, but complements evolution-aware RV. RTS re-runs a subset of tests in a new version, but evolution-aware RV re-monitors all tests against a subset of specs. Using JavaMOP to monitor RTS' output was shown to be slower than using $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ to monitor all tests against a subset of specs [52, 82]. One reason is that RTS re-incurs the full instrumentation cost, but $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$ save instrumentation costs on specs not re-monitored. Prior work also shows that combining RTS with evolution-aware RV is often faster than evolution-aware RV alone [52].

7 Conclusions

LazyMOP monitors fewer duplicate traces and is therefore faster than TraceMOP, the state-of-the-art online explicit-trace RV technique. To demonstrate the practical usefulness of fast explicit-trace RV, we also introduce LazyMOP^e, which reduces the generation of duplicate traces using traces from a prior program version. We show that the combination of LazyMOP^e with two state-of-the-art evolution-aware RV techniques is faster than these techniques alone in many projects. These results provide initial yet strong evidence that fast explicit-trace RV for software testing is practical and useful. We discuss other new trace-based RV applications that LazyMOP can enable and future work that can make LazyMOP and LazyMOP^e even faster.

Acknowledgments

We thank Saikat Dutta, Pengyue Jiang, Shinhae Kim, Sasa Misailovic, Elaine Yao, and the anonymous reviewers for their help, comments, and feedback. This work is partially supported by an Intel Rising Star Faculty Award, a Google Cyber NYC Institutional Research Award, and the United States National Science Foundation (NSF) under Grant Nos. CCF-2045596, CCF-2319473, CCF-2403035, CCF-2525243, CCF-2319472, and CCF-2349961.

Data-Availability Statement

Our formative study data, LazyMOP, LazyMOP^e, our scripts, and our experimental infrastructure are at https://github.com/SoftEngResearch/lazymop.

References

[1] Luca Aceto, Antonis Achilleos, Elli Anastasiadi, Adrian Francalanza, Daniele Gorla, and Jana Wagemaker. 2024. Centralized vs decentralized monitors for hyperproperties. *arXiv preprint arXiv:2405.12882* (2024).

[2] Abdulrahman Alshammari, Paul Ammann, Michael Hilton, and Jonathan Bell. 2024. 230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers. In *ICST*.

[3] ArtcatWebPage 2025. ARTCAT: Autonomic Response To Cyber-Attack. https://grammatech.github.io/prj/artcat.

[4] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, et al. 2005. Combining test case generation and runtime verification. *TCS* 336, 2-3 (2005).

[5] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser. 2003. Experiments with test case generation and runtime analysis. In *Abstract State Machines*.

[6] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. 2007. Statistical debugging using compound Boolean predicates. In *ISSTA*.

[7] ASMWebPage 2025. ASM. http://asm.ow2.org/.

[8] AsyncProfilerWebPage 2025. Sampling CPU and HEAP profiler for Java. https://github.com/async-profiler/async-profiler.

[9] Thomas Ball. 1998. On the Limit of Control Flow Analysis for Regression Test Selection. In *ISSTA*.

[10] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-Based Runtime Verification. In *VMCAI*.

[11] Howard Barringer, David Rydeheard, and Klaus Havelund. 2010. Rule Systems for Run-time Monitoring: From Eagle to RuleR. *Journal of Logic and Computation* 20, 3 (2010).

[12] David Basin, Felix Klaedtke, and Eugen Zălinescu. 2017. Runtime verification of temporal properties over out-of-order data streams. In *CAV*.

[13] David A Basin, Felix Klaedtke, and Eugen Zalinescu. 2017. The MonPoly Monitoring Tool. *RV-CuBES* 3 (2017).

[14] Omar Bataineh, David S Rosenblum, and Mark Reynolds. 2019. Efficient decentralized LTL monitoring framework using tableau technique. *TECS* 18, 5s (2019).

[15] Jon Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *ICSE*.

[16] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. 2024. Monitoring second-order hyperproperties. *arXiv preprint arXiv:2404.09652* (2024).

[17] John Bible, Gregg Rothermel, and David S. Rosenblum. 2001. A Comparative Study of Coarse- and Fine-grained Safe Regression Test-selection Techniques. *TOSEM* 10, 2 (2001).

[18] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. 2007. Collaborative Runtime Verification with Tracematches. In *RV*.

[19] Eric Bodden, Patrick Lam, and Laurie Hendren. 2008. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-time. In *FSE*.

[20] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. 2013. Time-triggered runtime verification. 43, 1 (2013).

[21] Lionel Briand, Yvan Labiche, and Siyuan He. 2009. Automating Regression Test Selection Based on UML Designs. *IST* 51, 1 (2009).

[22] BugInEmop 2024. Known bug in eMOP. https://github.com/SoftEngResearch/emop/issues/97.

[23] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. A survey of runtime monitoring instrumentation techniques. *arXiv preprint arXiv:1708.07229* (2017).

[24] Marek Chalupa and Thomas A Henzinger. 2023. Monitoring hyperproperties with prefix transducers. In *RV*.

[25] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. 2004. A Formal Monitoring-Based Framework for Software Development and Analysis. In *ICFEM*.

[26] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Roşu. 2009. Efficient formalism-independent monitoring of parametric properties. In *ASE*.

[27] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Roşu. 2009. *Efficient Formalism-Independent Monitoring of Parametric Properties.* Technical Report UIUCDCS-R-2009-11787. Computer Science Dept., UIUC.

[28] Feng Chen and Grigore Roşu. 2007. MOP: An efficient and generic runtime verification framework. In *OOPSLA*.

[29] Feng Chen and Grigore Roşu. 2003. Towards Monitoring-Oriented Programming: A paradigm combining specification and implementation. In *RV*.

[30] Feng Chen and Grigore Roşu. 2008. *Parametric trace slicing and monitoring.* Technical Report UIUCDCS-R-2008-2977. Computer Science Dept., UIUC.

[31] Feng Chen and Grigore Roşu. 2009. Parametric trace slicing and monitoring. In *TACAS*.

[32] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010).

[33] Christian Colombo, Gordon J Pace, and Patrick Abela. 2009. *Offline runtime verification with real-time properties: A case study.* Technical Report.

[34] Marcelo d'Amorim and Klaus Havelund. 2005. Event-based runtime verification of Java programs. In *WODA*.

[35] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning*.

[36] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2016. Runtime Monitoring with Union-Find Structures. In *TACAS*.

[37] Normann Decker, Martin Leucker, and Daniel Thoma. 2013. jUnit RV–adding runtime verification to jUnit. In *FM*.

[38] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. 2010. Runtime Verification in Context: Can Optimizing Error Detection Improve Fault Diagnosis?. In *RV*.

[39] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *IST* 52, 1 (2010).

[40] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical evaluations of regression test selection techniques: A systematic review. In *ESEM*.

[41] Ulfar Erlingsson and Fred B Schneider. 2000. IRM enforcement of Java stack inspection. In *IEEE S&P*.

[42] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2018. RVHyper: A runtime verification tool for temporal hyperproperties. In *TACAS*.

[43] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2019. Monitoring hyperproperties. *FMSD* 54, 3 (2019).

[44] Vojtěch Forejt, Marta Kwiatkowska, David Parker, Hongyang Qu, and Mateusz Ujma. 2012. Incremental Runtime Verification of Probabilistic Systems. In *RV*.

[45] Mark Gabel and Zhendong Su. 2012. Testing Mined Specifications. In *FSE*.

[46] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *ICSE Demo*.

[47] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *ISSTA*.

[48] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. 2014. Regression Test Selection for Distributed Software Histories. In *CAV*.

[49] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An empirical evaluation and comparison of manual and automated test selection. In *ASE*.

[50] Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. 2024. IronSpec: Increasing the Reliability of Formal Specifications. In *OSDI*.

[51] Kevin Guan and Owolabi Legunsen. 2024. An In-depth Study of Runtime Verification Overheads during Software Testing. In *ISSTA*.

[52] Kevin Guan and Owolabi Legunsen. 2025. Instrumentation-Driven Evolution-Aware Runtime Verification. In *ICSE*.

[53] Kevin Guan and Owolabi Legunsen. 2025. TraceMOP: An Explicit-Trace Runtime Verification Tool for Java. In *FSE Demo*.

[54] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *FSE Demo*.

[55] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *ISSRE*.

[56] Christopher Hahn. 2019. Algorithms for monitoring hyperproperties. In *RV*.

[57] Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2019. Constraint-based monitoring of hyperproperties. In *TACAS*.

[58] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *OOPSLA*.

[59] Negar Hashemi, Amjed Tahir, Shawn Rasheed, August Shi, and Rachel Blagojevic. 2025. Detecting and evaluating order-dependent flaky tests in JavaScript. In *ICST*.

[60] Matthias Hauswirth and Trishul M Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*.

[61] Klaus Havelund, Doron Peled, and Dogan Ulus. 2017. First order temporal logic monitoring with BDDs. In *FMCAD*.

[62] Klaus Havelund and Grigore Roşu. 2001. Monitoring Programs Using Rewriting. In *ASE*.
[63] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. 2014. Online monitoring of metric temporal logic. In *RV*.
[64] JaCoCoWebPage 2009. JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/.
[65] JavaForEach 2025. The For-Each Loop. https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html.
[66] JavaParserPage 2025. JavaParser - Home. https://javaparser.org.
[67] Omar Javed and Walter Binder. 2018. Large-Scale Evaluation of the Efficiency of Runtime-Verification Tools in the Wild. In *APSEC*.
[68] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. 2011. Garbage Collection for Monitoring Parametric Properties. In *PLDI*.
[69] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE Demo*.
[70] James A. Jones and Mary Jean Harrold. 2003. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *TSE* 29, 3 (2003).
[71] Hannes Kallwies, Martin Leucker, Malte Schmitz, Albert Schulz, Daniel Thoma, and Alexander Weiss. 2022. TeSSLa–an ecosystem for runtime verification. In *RV*.
[72] Henrik Karlsson. 2019. *Limiting Transitive Closure for Static Regression Test Selection approaches*. Master's thesis. KTH Royal Institute of Technology, Sweden.
[73] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An overview of AspectJ. In *ECOOP*.
[74] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 1999. Formally specified monitoring of temporal properties. In *ECRTS*.
[75] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*.
[76] Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. 2012. *Towards Categorizing and Formalizing the JDK API*. Technical Report. Computer Science Dept., UIUC.
[77] Owolabi Legunsen, Nader Al Awar, Xinyue Xu, Wajih Ul Hassan, Grigore Roşu, and Darko Marinov. 2019. How Effective are Existing Java API Specifications for Finding Bugs During Runtime Verification? *ASE Journal* 26, 4 (2019).
[78] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE*.
[79] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *ASE*.
[80] Owolabi Legunsen, Darko Marinov, and Grigore Roşu. 2015. Evolution-aware monitoring-oriented programming. In *ICSE NIER*.
[81] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *ASE*.
[82] Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, and Darko Marinov. 2019. Techniques for Evolution-Aware Runtime Verification. In *ICST*.
[83] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More precise regression test selection via reasoning about semantics-modifying changes. In *ISSTA*.
[84] Matt Luckcuck. 2020. Offline Runtime Verification of Safety Requirements using CSP. *arXiv preprint arXiv:2007.03522* (2020).
[85] Erik Lundsten. 2019. *EALRTS: A predictive regression test selection tool*. Master's thesis. KTH Royal Institute of Technology, Sweden.
[86] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.
[87] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *RV*.
[88] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *ICSE SEIP*.
[89] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI*.
[90] Michael Martin, Benjamin Livshits, and Monica S Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*.
[91] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. 2012. An overview of the MOP runtime verification framework. *IJSTTT* 14, 3 (2012).
[92] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing Runtime Verification Violations. In *ICST*.
[93] Samaneh Navabpour, Chun Wah Wallace Wu, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2011. Efficient Techniques for Near-Optimal Instrumentation in Time-Triggered Runtime Verification. In *RV*.

[94] OfficialJavaAgents 2014. java.lang.instrument. http://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html.
[95] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *TOSEM* 31, 1 (2021).
[96] Srinivas Pinisetty, Gerardo Schneider, and David Sands. 2018. Runtime verification of hyperproperties for deterministic programs. In *FMSD*.
[97] PITWebPage 2011. PIT Mutation Testing. http://pitest.org/.
[98] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2010. Monitor Optimization via Stutter-equivalent Loop Transformation. In *OOPSLA*.
[99] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2013. Optimizing Monitoring of Finite State Properties Through Monitor Compaction. In *ISSTA*.
[100] Shanto Rahman, Aaron Massey, Wing Lam, August Shi, and Jonathan Bell. 2024. Automatically reproducing timing-dependent flaky-test failures. In *ICST*.
[101] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA. In *TACAS*.
[102] Giles Reger and Klaus Havelund. 2016. What is a trace? A runtime verification perspective. In *International Symposium on Leveraging Applications of Formal Methods*.
[103] Adam Renberg. 2014. *Test-inspired runtime verification: Using a unit test-like specification syntax for runtime verification.* Master's thesis. KTH, Sweden.
[104] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *TSE* 39, 5 (2013).
[105] Grigore Roşu and Feng Chen. 2012. Semantics and algorithms for parametric monitoring. *LICS* 8 (2012).
[106] Gregg Rothermel and Mary Jean Harrold. 1993. A safe, efficient algorithm for regression test selection. In *ICSM*.
[107] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *TSE* 22, 8 (1996).
[108] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *TOSEM* 6, 2 (1997).
[109] Gregg Rothermel and Mary Jean Harrold. 1998. Empirical studies of a safe regression test selection technique. *TOSEM* 24, 6 (1998).
[110] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical Studies of Test-Suite Reduction. *STVR* 12, 4 (2002).
[111] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *FSE*.
[112] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *ICST*.
[113] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-suite Reduction in Real Software Evolution. In *ISSTA*.
[114] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. In *OOPSLA*.
[115] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *FSE*.
[116] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *FSE*.
[117] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *ISSRE*.
[118] Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell. 2024. The effects of computational resources on flaky tests. *TSE* 50, 12 (2024).
[119] Mats Skoglund and Per Runeson. 2005. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *ESEM*.
[120] Mats Skoglund and Per Runeson. 2007. Improving class firewall regression test selection by removing the class firewall. *JSEKE* 17, 3 (2007).
[121] Quinten David Soetens, Serge Demeyer, and Andy Zaidman. 2013. Change-Based Test Selection in the Presence of Developer Tests. In *CSMR*.
[122] Chukri Soueidi and Yliès Falcone. 2022. Residual runtime verification via reachability analysis. In *VSTTE*. 148–166.
[123] Robert E Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *TSE* 1 (1986).
[124] Leopoldo Teixeira, Breno Miranda, Henrique Rebêlo, and Marcelo d'Amorim. 2021. Demystifying the challenges of formally specifying API properties for runtime verification. In *ICST*.

[125] TraceMOPWeb 2024. TraceMOP: A Trace-Aware Runtime Verification Tool for Java. https://github.com/SoftEngResearch/tracemop.

[126] David Willmor and Suzanne M. Embury. 2005. A Safe Regression Test Selection Technique for Database Driven Applications. In *ICSM*.

[127] Chun Wah Wallace Wu, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2013. Reducing Monitoring Overhead by Integrating Event- and Time-Triggered Techniques. In *RV*.

[128] Guoqing Xu and Atanas Rountev. 2007. Regression test selection for AspectJ software. In *ICSE*.

[129] Ugur Yilmaz. 2019. *A Method for Selecting Regression Test Cases Based on Software Changes and Software Faults.* Master's thesis. Hacettepe University, Turkey.

[130] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2 (2012).

[131] Ayaka Yorihiro, Pengyue Jiang, Valeria Marques, Benjamin Carleton, and Owolabi Legunsen. 2023. eMOP: A Maven Plugin for Evolution-Aware Runtime Verification. In *RV*.

[132] Guofeng Zhang, Luyao Liu, Zhenbang Chen, and Ji Wang. 2024. Hybrid Regression Test Selection by Integrating File and Method Dependences. In *ASE*.

[133] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and combining analysis-based and learning-based regression test selection. In *AST*.

[134] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *ICSE*.

[135] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *ISSRE*.

[136] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *ICSE*.

Received 2025-03-26; accepted 2025-08-12